

# FRANZ LISP

## Reference Manual

*Opus 43*  
March 1987

210-114-2

This revision incorporates minor corrections and changes to the University of California text, major new sections on packages, flavors, and the top-level environment, and Common LISP compatibility features.

Portions ©1980, 1981, 1983 by the Regents of the University of California. Additional ©1984, 1985, 1986, 1987 by Franz Incorporated, Alameda, California. All rights reserved.

<i>Original text</i>	John K. Foderaro Keith L. Sklower Kevin Layer
<i>Opus 42 text</i>	Kevin Layer Richard J. Fateman Steven M. Haflich
<i>Opus 43 changes</i>	Charles A. Cox
<i>Typography</i>	Harry Weeks
<i>Production</i>	Tomi Izuno



# Contents





# Contents

## Preface

*A short overview and history of FRANZ LISP. Comments, suggestions, reporting bugs.*

## 1 Introduction

*Introduction to FRANZ LISP, and description of data types and notation.*

- 1.1 Data types 1-1
- 1.2 Documentation conventions 1-6

## 2 Data structure access

*Functions to create, destroy, and manipulate LISP data objects.*

- 2.1 Lists 2-1
- 2.2 Predicates 2-8
- 2.3 Symbols and strings 2-12
- 2.4 Vectors 2-21
- 2.5 Hash tables 2-24
- 2.6 Arrays 2-27
- 2.7 Hunks 2-30
- 2.8 Bcds 2-31
- 2.9 Structures 2-31
- 2.10 Random functions 2-35

## 3 Arithmetic and logic functions

*Functions that perform arithmetic and logic operations.*

- 3.1 Simple arithmetic functions 3-1
- 3.2 Predicates 3-3
- 3.3 Trigonometric functions 3-4
- 3.4 Bignum and fixnum manipulation 3-5
- 3.5 Bit manipulation 3-6
- 3.6 Other functions 3-8

## 4 Special functions

*Functions that alter the flow of control and map other functions over lists.*

4.1 Introduction 4-1

4.2 Functions 4-1

4.3 Multiple value returns 4-20

## 5 Input/output

*Functions to read from, and write to, ports, and functions that modify reader syntax.*

5.1 Introduction 5-1

5.2 Functions 5-2

5.3 Format 5-11

## 6 System functions

*Functions for storage management, debugging, and reading and setting global LISP status variables. Description of operating-system interface functions.*

6.1 System functions 6-1

6.2 The Lisp Profiler 6-13

## 7 The reader

*The LISP reader, character macros, and syntax codes.*

7.1 Introduction 7-1

7.2 Syntax classes 7-1

7.3 Reader operations 7-2

7.4 Character classes 7-3

7.5 Syntax classes 7-6

7.6 Character macros 7-10

7.7 Functions 7-12

## 8 Program forms

*Special forms (including macros), functional objects, and the backquote character macro.*

8.1 Valid function objects 8-1

8.2 Functions 8-1

8.3 Macros 8-6

8.4 Closures and fclosures 8-12

8.5 Functional objects table 8-15



## 9 Arrays and vectors

*Detailed description of MacLISP-compatible arrays and of array internals.*

- 9.1 General arrays 9-1
- 9.2 Subparts of an array object 9-2
- 9.3 The MacLISP-compatible array package 9-3
- 9.4 Vectors 9-4
- 9.5 Anatomy of vectors 9-4
- 9.6 Immediate vectors 9-5

## 10 Exception handling

*Anatomy of an error; autoloading.*

- 10.1 Errset and error handler functions 10-1
- 10.2 The anatomy of an error 10-1
- 10.3 Error handling algorithm 10-2
- 10.4 Default aids 10-3
- 10.5 Autoloading 10-3
- 10.6 Interrupt processing 10-3

## 11 The Lister trace package

*Description of a very useful debugging aid.*

- 11.1 The trace function 11-1
- 11.2 Trace special variables 11-4
- 11.3 Other trace package functions 11-5

## 12 Liszt, the LISP compiler

*Use of the compiler and hints for making functions compilable.*

- 12.1 General strategy of the compiler 12-1
- 12.2 Running the compiler 12-1
- 12.3 Special forms 12-1
- 12.4 Using the compiler 12-4
- 12.5 Compiler options 12-5
- 12.6 Autorun 12-7
- 12.7 Pure literals 12-8
- 12.8 Transfer tables 12-8
- 12.9 Fixnum functions 12-9

## 13 The top level

*Description of FRANZ LISP's top level which includes access to debugging tools, a history mechanism, and execution stepper.*

- 13.1 Introduction 13-1

- 13.2 A top-level for debugging LISP programs 13-1
- 13.3 How to use tpl 13-1
- 13.4 Tpl special symbols 13-6
- 13.5 Sample sessions with tpl 13-7
- 13.6 The file subsystem 13-12
- 13.7 File subsystem implementation notes 13-14

## 14 Structured data types

*Exposition on the **defstruct** facility for defining and using composite data.*

- 14.1 Introduction 14-1
- 14.2 Setf and defsetf 14-1
- 14.3 Defstruct 14-4

## 15 The LISP stepper and Fixit

*Programs that permit single-stepping through a LISP program, and examining and modifying the evaluation stack to fix bugs on the fly.*

- 15.1 Simple use of stepping 15-1
- 15.2 Advanced features 15-2
- 15.3 Overhead of stepping 15-3
- 15.4 Evalhook and funcallhook 15-3
- 15.5 The fixit debugger 15-4

## 16 The LISP editor

*A structure editor for interactive modification of FRANZ LISP code.*

- 16.1 Introduction 16-1
- 16.2 Tutorial 16-1
- 16.3 Editor functions 16-5
- 16.4 Printing commands 16-7
- 16.5 Scope of attention 16-8
- 16.6 Pattern and search commands 16-10
- 16.7 Location specifications 16-12
- 16.8 Structure modification commands 16-13
- 16.9 List modification commands 16-16
- 16.10 Undoing commands 16-18
- 16.11 Commands that evaluate 16-19
- 16.12 Commands that test 16-20
- 16.13 Editor macros 16-20
- 16.14 Miscellaneous editor commands 16-22



## 17 Packages

*Description of modular organization techniques for LISP.*

- 17.1 Introduction 17-1
- 17.2 Package names 17-2
- 17.3 Translating strings to symbols 17-3
- 17.4 Exporting and importing symbols 17-3
- 17.5 Built-in packages 17-4
- 17.6 Package system functions and variables 17-4
- 17.7 Modules 17-9
- 17.8 Creating packages in files 17-10

## 18 The foreign function interface

*Mechanics of the interface between FRANZ LISP and other languages.*

- 18.1 A simple example 18-1
- 18.2 Loading C functions 18-2
- 18.3 Foreign-function disciplines 18-4
- 18.4 Calling C functions 18-5
- 18.5 The C program 18-16
- 18.6 Fortran and Pascal 18-18
- 18.7 Pipes 18-22

## 19 Objects, message-passing, and flavors

*Object-oriented programming techniques in FRANZ LISP.*

- 19.1 Introduction 19-1
- 19.2 Objects 19-1
- 19.3 Modularity 19-3
- 19.4 Generic operations 19-6
- 19.5 Generic operations in LISP 19-7
- 19.6 Simple use of flavors 19-9
- 19.7 Mixing flavors 19-12
- 19.8 Flavor functions 19-16
- 19.9 Defflavor options 19-24
- 19.10 Flavor families 19-32
- 19.11 Vanilla flavor 19-32
- 19.12 Method combination 19-34
- 19.13 Implementation of flavors 19-42
- 19.14 Property list operations 19-45
- 19.15 Copying instances 19-46

**A Summary of FRANZ LISP symbols**

*An index and summary for FRANZ LISP functions, variables, and keywords.*

**B List of special symbols**

*Symbols with special meanings to the LISP system.*

**C The garbage collector**

*A short exposition on dynamic memory allocation in FRANZ LISP.*

**D Lxref**

*The LISP cross-reference program.*

**E Reconfiguring lisp**

*How to reconfigure LISP from the distribution files.*



# Preface

This reference manual describes FRANZ LISP, a popular dialect of LISP that runs on a wide spectrum of computers. FRANZ LISP is a traditional dynamically-scoped LISP of the MacLISP family. Originally developed at the University of California at Berkeley, it has been greatly enhanced and optimized by Franz Incorporated.

This manual describes FRANZ LISP in detail and assumes the reader is familiar with LISP. It does not attempt to be a tutorial for the LISP language. An excellent book, *LISP-craft* by Robert Wilensky (W. W. Norton), is recommended as a tutorial introduction to FRANZ LISP.

## The language

The FRANZ LISP system consists of an interpreter, *lisp*, and an optimizing compiler, *liszt*. FRANZ LISP supports an efficient and robust implementation of Flavors, a package system compatible with Common LISP, a user-friendly top level with access to extensive debugging facilities, a foreign-function interface, multiple-valued returns, user-defined data structures, a parameterized reader, keyword arguments, and a rich collection of data types including hash tables and generalized arrays. A large number of Common LISP compatibility functions are implemented, as well as compatibility modes for MacLISP, UCILISP, and Lisp Machine LISP (ZetaLISP and Symbolics LISP).

The kernel of FRANZ LISP is written almost entirely in the programming language C, with much of the support written in (compiled) LISP. For run-time efficiency, small portions of the kernel are written in assembly language.

## History

In the late 60's and early 70's, a group of researchers at MIT developed a symbolic algebraic system, called Macsyma. Macsyma is a very large program (about 300,000 lines of source code), written entirely in MacLISP, and even today remains one of the largest LISP program ever written. Several years after relocating from MIT to the Electrical Engineering and Computer Sciences Department at UC Berkeley, Richard Fateman, one of the original contributors to Macsyma, saw that the DEC VAX-11/780 with a large address space, could be an excellent system to allow Macsyma to grow even larger. For various reasons, it became important to use the Unix operating system on the first VAX at Berkeley, and a paging version was implemented. Fateman then set out to make a LISP which would run Macsyma. The first considerations were that it be MacLISP



compatible, as small as possible, fast and capable of providing the huge address space that the PDP-10 MacLISP lacked. Some students and two researchers from Bell Labs, then developed the first version of FRANZ LISP, in 1978. Subsequently, FRANZ LISP was released with Berkeley UNIX, and today more than 20,000 sites are using FRANZ LISP.

After Macsyma (or Vaxima, as the VAX version of Macsyma was called) was running on the VAX, FRANZ LISP started to take new directions. Because of the Berkeley Artificial Intelligence Group (BAIR) headed by Robert Wilensky, numerous sites which received FRANZ LISP on UNIX tapes, and a voracious Berkeley user community, FRANZ LISP started to grow into a LISP system meant to be used for serious research and development of other systems. In this period, such programs as OPS-5, PEARL and many others were written in FRANZ LISP.

In 1984, Franz Inc. was founded through the cooperation of the members of the FRANZ LISP team at UC Berkeley, and this manual describes the culminations of several years of enhancements, bug fixes, and development on FRANZ LISP.

## Comments and suggestions

We are always seeking a dialogue with our users to find ways of improving FRANZ LISP. We invite your comments and suggestions. A form is provided at the back of this manual for your convenience, but of course a personal letter is always welcome. There is an electronic mail forum called [franz-friends@Berkeley.EDU](mailto:franz-friends@Berkeley.EDU) for Franz users where items of general interest are often posted. We encourage you to join, by sending your address to [franz-friends-request@Berkeley.EDU](mailto:franz-friends-request@Berkeley.EDU)

## Reporting bugs

The technical staff at Franz Incorporated is committed to the highest standards of software engineering. Releases of FRANZ LISP are extensively tested both internally and in the field before wide dissemination. Nonetheless, it is possible that you will find bugs or encounter behavior that you might not have expected. In this event, Franz Incorporated will do its utmost to resolve the problem. However, resolving bugs is a cooperative venture, and we need your help.

Before reporting a bug, please study this manual to be sure that what you experienced was indeed a bug. If the documentation is not clear, this is a bug in the documentation! FRANZ LISP may not have done what you expected, but it might have been doing what it was supposed to do. In the latter case, you may feel strongly that the documented behavior is incorrect. We will carefully consider your point of view. If you feel the manual and the behavior are wrong, we may consider altering both in a future release. A report that such and such happened is generally of limited value in determining the cause of the problem. It is very important for us to know what happened before the error occurred: what you typed in, what FRANZ LISP typed out. If you are able to localize the bug and reliably duplicate it with a minimal amount of code, it will greatly expedite repairs. It's much easier to find a bug that is generated when a single isolated function is applied than a bug that is generated *somewhere* when an enormous application is loaded.



Although we are more familiar with FRANZ LISP, at least internally, than you may be, you are certainly more familiar with your application, and the context in which the bug was observed.

Context is also important in determining whether the bug is really in FRANZ LISP or in something that it depends on, e.g. the operating system. We suggest you gather the following bits of information before submitting the bug report to us:

- The version number of the FRANZ LISP interpreter, as printed on the herald.
- The version number of the liszt compiler, as printed on its herald.
- The site name, machine type, operating system including version number.
- A literatim log, if possible, starting from a fresh execution of LISP.

We will meet you more than half way to get your project moving again when a bug stalls you. We ask only that you take a few steps in our direction.

You may reach our Customer Service Department at (415) 548-3600. Or you may write to us at Franz Incorporated, 1995 University Avenue, Berkeley, California 94704. You may also send electronic mail to:

**...!ucbvax!franz!franz-bugs**

via UUCP, or to:

**franz-bugs%franz.uucp@Berkeley.EDU**

via ARPAnet.



100

100

1. The first part of the report is a general introduction to the subject of the study.

2. The second part of the report is a detailed description of the methods used in the study.

3. The third part of the report is a discussion of the results of the study.

4. The fourth part of the report is a conclusion and a list of references.

5. The fifth part of the report is a list of appendices.

6. The sixth part of the report is a list of figures and tables.

7. The seventh part of the report is a list of footnotes.

8. The eighth part of the report is a list of abbreviations.

9. The ninth part of the report is a list of symbols.

10. The tenth part of the report is a list of units.

11. The eleventh part of the report is a list of definitions.

12. The twelfth part of the report is a list of acknowledgments.

13. The thirteenth part of the report is a list of references.

14. The fourteenth part of the report is a list of appendices.

15. The fifteenth part of the report is a list of figures and tables.

16. The sixteenth part of the report is a list of footnotes.

17. The seventeenth part of the report is a list of abbreviations.

# 1 Introduction

1.1	Data types	1-1
1.1.1	Lispval	1-1
1.1.2	Symbol	1-1
1.1.3	List	1-2
1.1.4	Binary	1-2
1.1.5	Fixnum	1-3
1.1.6	Flonum	1-3
1.1.7	Bignum	1-3
1.1.8	String	1-3
1.1.9	Port	1-3
1.1.10	Vector	1-4
1.1.11	Array	1-4
1.1.12	Value	1-4
1.1.13	Hunk	1-5
1.1.14	Package	1-5
1.1.15	Closure	1-5
1.1.16	Hash tables	1-5
1.1.17	Other	1-5
1.2	Documentation conventions	1-6





# 1 Introduction

This document is a reference manual for the FRANZ LISP system. It is not a LISP primer nor an introduction to the language. It assumes that you are familiar with at least one dialect of LISP, preferably a member of the MacLISP or Common LISP family.

A recommended text for learning LISP, with specific reference to FRANZ LISP is *LISPCraft* by Robert Wilensky, published by W. W. Norton (1984).

This chapter describes the data types of FRANZ LISP and the conventions used in the description of the FRANZ LISP functions. In an attempt to be concise, we use a shorthand given in §1.2. It is very important that these conventions be read for a full understanding of subsequent sections. You should refer to the table in that section if the data types of function arguments are in question.

## 1.1 Data types

FRANZ LISP has a collection of data types for system implementation and programming. This section describes each type briefly, and, if a type is divisible, the insides are examined. There is a LISP function **type** that returns the type name of a LISP object. This is the official FRANZ LISP name for that type, and this name only is used in the manual to avoid confusing you. The types are listed in terms of importance rather than alphabetically.

### 1.1.1 Lispval

This is the name used to describe any LISP object. The function **type** never returns *lispval*.

### 1.1.2 Symbol

This object corresponds to a variable in most other programming languages. It may have a value or may be *unbound*. A symbol may be *lambda bound* meaning that its value (conceptually, at least) is pushed on a stack, and the symbol is given a new value for the duration of a certain context. When the LISP processor leaves that context, the symbol's value is popped off the stack.

A symbol may also have a *function binding*. This function binding is static; it cannot be *lambda bound*. Whenever the symbol is used in the functional position of a LISP expression the function binding of the symbol is examined. See Chapter 4 for more



details on evaluation.

A symbol may also have a *property list*, another static data structure. The property list is a list of an even number of elements, considered to be grouped as pairs. The first element of the pair is the *indicator*; the second, the *value* of that indicator.

Each symbol has a print name, *pname*, which is how this symbol is identified from input and referred to on (printed) output.

The function **intern** is the usual way of creating a symbol, but the functions **concat**, **maknam**, and their derivatives also create symbols. Usually, symbols are a member of a package, but not always. (See chapter 17 for a description of packages.)

<i>Subpart name</i>	<i>Get value</i>	<i>Set value</i>	<i>Type</i>
value	<b>eval</b>	<b>set</b> <b>setq</b>	lispval
property list	<b>plist</b> <b>get</b>	<b>setplist</b> <b>putprop</b> <b>defprop</b>	list, nil
function binding	<b>getd</b>	<b>putd</b> <b>def</b>	array, binary, list, nil
print name	<b>get_pname</b>		string
home package	<b>symbol-package</b>		package

### 1.1.3 List

A list cell has two parts, called the *car* and *cdr*. List cells are created by the function **cons**.

<i>Subpart name</i>	<i>Get value</i>	<i>Set value</i>	<i>Type</i>
<b>car</b>	<b>car</b>	<b>rplaca</b>	lispval
<b>cdr</b>	<b>cdr</b>	<b>rplacd</b>	lispval

### 1.1.4 Binary

This type acts as a function header for machine coded functions. It has two parts: a pointer to the start of the function and a symbol whose print name describes the argument *discipline*. The discipline (if **lambda**, **macro**, or **nlambda**) determines whether the arguments to this function are evaluated by the caller before this function is called. If the discipline is a string (specifically *"subroutine"*, *"function"*, *"integer-function"*, *"real-function"*, *"c-function"*, *"double-c-function"*, *"void-c-function"*, or *"vector-c-function"*) then this function is a foreign subroutine or function. (See §8.5 for more details on this.) Although the type of the *entry* field of a binary type object is usually *string* or *other*, the object pointed to is actually a sequence of machine instructions.



Objects of type binary are created by **mfunction**, **cfasl**, and **getaddress**.

<i>Subpart name</i>	<i>Get value</i>	<i>Set value</i>	<i>Type</i>
entry	<b>getentry</b>		string, fixnum
discipline	<b>getdisc</b>	<b>putdisc</b>	symbol, fixnum

### 1.1.5 Fixnum

A fixnum is an integer constant in the range  $-(2^{30}-1)$  to  $2^{30}-1$ . Small fixnums ( $-1024$  to  $1023$ ) are stored in a special table so they need not be allocated each time one is needed.

### 1.1.6 Flonum

A flonum is a double-precision real number.

### 1.1.7 Bignum

A bignum is an integer of potentially unbounded size. When integer arithmetic exceeds the limits of fixnums mentioned above, the calculation is automatically done with bignums. If calculation with bignums gives a result that can be represented as a fixnum, then the fixnum representation is used.<sup>1</sup> This contraction is known as *integer normalization*. Many LISP functions assume that integers are normalized. Bignums are composed of a sequence of list cells and a cell known as an *sdot*. You should consider a bignum structure indivisible and use functions such as **halpart** and **bignum-leftshift** to extract parts of it.

### 1.1.8 String

A string is a null terminated sequence of characters. Most functions of symbols that operate on the symbol's print name also work on strings. The default reader syntax is set so that a sequence of characters surrounded by double quotes is a string.

### 1.1.9 Port

A port is a structure that the system I/O routines can reference to transfer data between the LISP system and external media. Unlike other LISP objects there are a very limited number of ports (20 on most machines, more on others). Ports are allocated by **infile** and **outfile** and deallocated by **close** and **resetio**. During initialization, FRANZ LISP binds the symbol **piport** to a port attached to the standard input stream. This port prints as **#<port stdin>**. There are ports connected to the standard output and error streams, which

<sup>1</sup>In certain cases, the current algorithms for integer arithmetic operations return a result as a bignum even though it could just barely be represented as a fixnum.



print as `#<port stdout>` and `#<port stderr>`. This is discussed in more detail at the beginning of Chapter 5.

### 1.1.10 Vector

Vectors are indexed sequences of data. They can be used to implement a notion of user-defined types via their associated property list. They make hunks (see below) logically unnecessary, except for compatibility reasons. There is a second kind of vector, called an immediate-vector, that stores binary data. The name that the function **type** returns for immediate-vectors is **vectori**. For example, immediate-vectors can be used to implement strings and block-flonum arrays. Vectors are discussed in chapter 9. The functions **new-vector** and **vector** can be used to create vectors.

<i>Subpart name</i>	<i>Get value</i>	<i>Set value</i>	<i>Type</i>
<b>datum[i]</b>	<b>vref</b>	<b>vset</b>	<b>lispval</b>
<b>property</b>	<b>vprop</b>	<b>vsetprop</b> <b>vputprop</b>	<b>lispval</b>
<b>size</b>	<b>vsize</b>		<b>fixnum</b>

### 1.1.11 Array

Arrays are rather complicated types and are fully described in Chapter 9. An array consists of a block of contiguous data, a function to access that data, and auxiliary fields for use by the accessing function. Since an array's accessing function is created by the user, you can create the array to have any form you choose (e.g. n-dimensional, triangular, or hash table).

Arrays are created by the function **marray**.

<i>Subpart name</i>	<i>Get value</i>	<i>Set value</i>	<i>Type</i>
<b>access function</b>	<b>getaccess</b>	<b>putaccess</b>	<b>binary, list, symbol</b>
<b>auxiliary</b>	<b>getaux</b>	<b>putaux</b>	<b>lispval</b>
<b>data</b>	<b>arrayref</b>	<b>replace set</b>	<b>block of contiguous lispval</b>
<b>length</b>	<b>getlength</b>	<b>putlength</b>	<b>fixnum</b>
<b>delta</b>	<b>getdelta</b>	<b>putdelta</b>	<b>fixnum</b>

### 1.1.12 Value

A value cell contains a pointer to a lispval. This type is used mainly by arrays of general LISP objects. Value cells are created with the **ptr** function. A value cell containing a



pointer to the symbol foo is printed as *(ptr to)foo*.

### 1.1.13 Hunk

A hunk is a vector of from 1 to 128 lispvals. Once a hunk is created (by **hunk** or **makhunk**) it cannot grow or shrink. The access time for an element of a hunk is slower than that for a list cell element but faster than for an array element. Hunks are really only allocated in sizes that are powers of two, but can appear to you to be any size in the 1 to 128 range. Realize that (not (atom lispval)) returns true if lispval is a hunk. Most LISP systems do not have a direct test for a list cell, and instead use the above test and assume that a true result means lispval is a list cell. In FRANZ LISP, you can use **dtpr** (dotted pair) to check for a list cell. Although hunks are not list cells, you can still access the first two hunk elements with **cdr** and **car**, and you can access any hunk element with **cxr**.<sup>2</sup> You can set the value of the first two elements of a hunk with **rplacd** and **rplaca** and you can set the value of any element of the hunk with **rplacx**. A hunk is printed by printing its contents surrounded by { and }. However, a hunk cannot be read in this way in the standard LISP system. It is easy to write a reader macro to do this if desired.

### 1.1.14 Package

Chapter 17 describes the FRANZ LISP package system. The ideas, adopted from Common LISP, are helpful for the development of large systems in a modular fashion.

### 1.1.15 Closure

A closure is a functional object designed to hold bindings of *special* variables between invocations. They are described in chapter 8.

### 1.1.16 Hash tables

Hash tables are LISP objects that efficiently map a LISP object onto another associated LISP object. Hash tables contain a set of entries that each associate a LISP object called the *key* with another LISP object called the *value*. Given the *key*, the corresponding *value* can be retrieved very rapidly.

### 1.1.17 Other

Occasionally, you can obtain a pointer to storage not allocated by the LISP system. One example of this is the entry field of those FRANZ LISP functions written in C. Such objects are classified as of type other. Foreign functions, which call **malloc()** to allocate their own space, may also inadvertently create such objects. The garbage collector ignores them.

<sup>2</sup>In a hunk, the function **cdr** references the first element and **car** the second.



## 1.2 Documentation conventions

The conventions used in the following chapters are designed to give a great deal of information in a brief space. The first line of a function description contains the function name in **bold face gothic**<sup>3</sup> and then lists the arguments in gothic, if there are any. The arguments all have names that begin with a letter or letters and an underscore. The initial letter or letters give the allowable type or types for that argument according to the table below.

Letter	Allowable type(s)
<b>g</b>	any type
<b>s</b>	symbol (although <i>nil</i> may not be allowed)
<b>t</b>	string
<b>l</b>	list (although <i>nil</i> may be allowed)
<b>n</b>	number (fixnum, flonum, bignum)
<b>i</b>	integer (fixnum, bignum)
<b>x</b>	fixnum
<b>b</b>	bignum
<b>f</b>	flonum
<b>u</b>	function type (either binary or lambda body)
<b>y</b>	binary
<b>v</b>	vector
<b>V</b>	vectori
<b>H</b>	hash table
<b>a</b>	array
<b>e</b>	value
<b>p</b>	port (or <i>nil</i> )
<b>h</b>	hunk
<b>k</b>	package
<b>c</b>	closure

In the first line of a function description we provide a template showing you how to call the function from the top level, including appropriate quote marks. (In FRANZ LISP, the form 'foo is the same as (quote foo).) For example, **cons** could be described via (cons 'g\_first 'g\_rest), allowing us to say the result is the list cell x where (car x) = g\_first, and (cdr x) = g\_rest. Referring to the previous table, we see that the 'g' prefix means 'general' or 'any type.' The suffixes first and rest are chosen to be mnemonically useful. Those arguments preceded by a quote mark are evaluated before the function is applied. This allows us to refer to the result of the **cons** without saying "the result of evaluating

<sup>3</sup>Note that gothic is a sans-serif type (i.e. no little tails on the letters), while the general typeface has serifs.



its first argument . . . ,” and so on. Occasionally a function will utilize some special order of evaluation (or non-evaluation) but if the arguments are generally evaluated, we will still use this notation.

When an argument is not quoted in the function description line, it is usually because no evaluation is done on that argument. Rarely, however, the argument is evaluated but at a time specifically mentioned in the function description.

Optional arguments are surrounded by square brackets. An ellipsis ( . . . ) means zero or more occurrences of an argument of the directly preceding type.

In the main text different type faces are used to distinguish between functions, symbols, constants, printed forms, and examples. Functions are printed in **bold gothic**. Other symbols are printed in gothic. Constants (such as 0, #A, "A", or nil) and special symbols (such as \*package\*) are printed in *italic gothic*. Keywords and lambda-list signifiers (such as :test and &optional, respectively) are indicated in **bold italic gothic**. Printed forms are printed in *italic gothic*. Examples in the text are printed in gothic. When examples appear separate from the text, output from the FRANZ LISP system is printed in courier; input to FRANZ LISP is printed in **bold courier**; and comments are printed in *italic courier*.





## 2 Data structure access

- 2.1 Lists 2-1
  - 2.1.1 List creation 2-1
  - 2.1.2 List predicates 2-3
  - 2.1.3 List accessing 2-4
  - 2.1.4 List manipulation 2-5
- 2.2 Predicates 2-8
- 2.3 Symbols and strings 2-12
  - 2.3.1 Symbol and string creation 2-12
    - 2.3.1.1 Symbol generating facility 2-14
  - 2.3.2 String and symbol predicates 2-16
  - 2.3.3 Symbol and string accessing 2-17
  - 2.3.4 Symbol and string manipulation 2-19
- 2.4 Vectors 2-21
  - 2.4.1 Vector creation 2-21
  - 2.4.2 Vector predicate 2-22
  - 2.4.3 Vector reference 2-22
  - 2.4.4 Vector modification 2-23
  - 2.4.5 Miscellaneous 2-23
- 2.5 Hash tables 2-24
  - 2.5.1 Hash table functions 2-24
- 2.6 Arrays 2-27
  - 2.6.1 Array creation 2-27
  - 2.6.2 Array predicate 2-28
  - 2.6.3 Array accessors 2-28
  - 2.6.4 Array manipulation 2-29
- 2.7 Hunks 2-30
  - 2.7.1 Hunk creation 2-30
  - 2.7.2 Hunk accessor 2-30
  - 2.7.3 Hunk manipulators 2-31
- 2.8 Bcds 2-31
- 2.9 Structures 2-31
  - 2.9.1 Association lists 2-31
  - 2.9.2 Property list 2-32
  - 2.9.3 Tconc structure 2-34
- 2.10 Random functions 2-35





## 2 Data structure access

The functions described in this chapter allow you to create and manipulate the various types of lisp data structures. Refer to §1.2 for a brief overview of the data structures available in FRANZ LISP.

### 2.1 Lists

The following functions exist for the creation and manipulation of lists. Lists are composed of a linked list of objects. Various authors call these either *list cells*, *cons cells*, or *dtptr cells*. Lists are normally terminated with the special symbol *nil*. *nil* is both a symbol and a representation for the empty list ().

#### 2.1.1 List creation

(cons 'g\_arg1 'g\_arg2)

- RETURNS a new list cell whose **car** is **g\_arg1** and whose **cdr** is **g\_arg2**.

(xcons 'g\_arg1 'g\_arg2)

- EQUIVALENT TO: (cons 'g\_arg2 'g\_arg1).

(ncons 'g\_arg)

- EQUIVALENT TO: (cons 'g\_arg nil).

(list ['g\_arg1 ... ])

(list\* ['g\_arg1 ... ])

- RETURNS a list whose elements are the **g\_arg*i***.
- NOTE: Function **list\*** differs from **list** in that the last argument is **consed** on to the list.
- EXAMPLE: (list 'x 'y '(z w)) is (x y (z w)) and (list\* 'x 'y '(z w)) is (x y z w).

(append 'l\_arg1 'l\_arg2 [ ... ])

- RETURNS a list containing the elements **l\_arg1** followed by **l\_arg2**.
- NOTE: To generate the result, the top level list cells of **l\_arg1** are duplicated and the **cdr** of the last list cell is set to point to **l\_arg2**. Thus this is an expensive



operation if `l_arg1` is large. See the descriptions of `nconc` and `tconc` for cheaper ways of doing the `append` if the list `l_arg1` can be altered.

`(append1 'l_arg1 'g_arg2)`

- RETURNS a list like `l_arg1` with `g_arg2` as the last element.
- NOTE: This is equivalent to `(append 'l_arg1 (list 'g_arg2))`.

---

```
; A common mistake is using append to add  
; one element to the end of a list.  
=> (append '(a b c d) 'e)  
(a b c d . e)  
; The user intended to say:  
=> (append '(a b c d) '(e))  
(a b c d e)  
; Better yet is append1.  
=> (append1 '(a b c d) 'e)  
(a b c d e)
```

---

`(quote! [g_qform1] ... [! 'g_iform1] ... [!! 'l_form1] ...)`

- RETURNS the list resulting from the splicing and insertion process described below.

■ NOTE: `quote!` is the complement of the `list` function. `list` forms a list by evaluating each form in the argument list; evaluation is suppressed if the form is `quoted`. In `quote!`, each form is implicitly `quoted`. To be evaluated, a form must be preceded by one of the evaluation operators `!` or `!!`. `! g_iform` evaluates `g_iform` and the value is inserted in the place of the call; `!! l_form` evaluates `l_form` and the value is spliced into the place of the call.

□ 'Splicing in' means that the parentheses surrounding the list are removed as the example below shows. Use of the evaluation operators can occur at any level in a form argument.

□ Another way to get the effect of the `quote!` function is to use the backquote character macro (see § 8.3.4).

---

```
(quote! cons ! (cons 1 2) 3) = (cons (1 . 2) 3)  
(quote! 1 !! (list 2 3 4) 5) = (1 2 3 4 5)  
(quote! try ! '(this ! one)) = (try (this ! one))
```

---

**(bignum-to-list 'b\_arg)**

- RETURNS a list of the fixnums which are used to represent the bignum.
- NOTE: The inverse of this function is **list-to-bignum**.

**(list-to-bignum 'l\_ints)**

- WHERE *l\_ints* is a list of fixnums.
- RETURNS a bignum constructed of the given fixnums.
- NOTE: The inverse of this function is **bignum-to-list**.

**2.1.2 List predicates****(dtptr 'g\_arg)**

- RETURNS *t* if *g\_arg* is a list cell.
- NOTE: (dtptr '()) is *nil*. The name **dtptr** is a contraction for "dotted pair."

**(listp 'g\_arg)**

- RETURNS *t* if *g\_arg* is a list object or *nil*.

**(tailp 'l\_x 'l\_y)**

- RETURNS *l\_x*, if a list cell **eq** to *l\_x* is found by **cdring** down *l\_y* zero or more times; *nil* otherwise.

---

```
=> (setq x '(a b c d) y (cddr x))
(c d)
; x and y are dotted pairs and lists
=> (and (dtptr x) (listp x))
t
; () is the same as nil and is not a dtptr
=> (dtptr '())
nil
; however, it is a list
=> (listp '())
t
=> (tailp y x)
(c d)
```

---

**(length 'l\_arg)**

- RETURNS the number of elements in the top level of list *l\_arg*.



### 2.1.3 List accessing

(car 'l\_arg)

(cdr 'l\_arg)

■ RETURNS the appropriate part of a **cons** cell. (car (cons x y)) is always x, (cdr (cons x y)) is always y. In FRANZ LISP, the **cdr** portion is located first in memory. This is hardly noticeable, and we mention it primarily as a curiosity.

(c..r 'lh\_arg)

■ WHERE the .. represents any positive number of a's and d's.

■ RETURNS the result of accessing the list structure in the way determined by the function name. The a's and d's are read from right to left, a d directing the access down the **cdr** part of the list cell and an a down the **car** part.

■ NOTE: lh\_arg may also be *nil*, and it is guaranteed that the **car** and **cdr** of *nil* is *nil*. If lh\_arg is a hunk, then (car 'lh\_arg) is the same as (cxr 1 'lh\_arg) and (cdr 'lh\_arg) is the same as (cxr 0 'lh\_arg).

It is generally hard to read and understand the context of functions with large strings of a's and d's, but these functions are supported by rapid accessing and open-compiling (see Chapter 12).

(nth 'x\_index 'l\_list)

■ RETURNS The nth element of l\_list, assuming zero-based index. Thus (nth 0 l\_list) is the same as (car l\_list). nth is both a function and a compiler macro so that more efficient code might be generated than for nthelem (described below).

■ NOTE: If x\_arg1 is nonpositive or greater than the length of the list, *nil* is returned.

(nthcdr 'x\_index 'l\_list)

■ RETURNS the result of **cdring** down the list l\_list x\_index times.

■ NOTE: If x\_index is less than 0, then (cons nil 'l\_list) is returned.

(nthelem 'x\_arg1 'l\_arg2)

■ RETURNS The x\_arg1st element of the list l\_arg2.

■ NOTE: This somewhat nonstandard name of this function comes from the PDP-11 LISP system.

(last 'l\_arg)

■ RETURNS The last list cell in the list l\_arg.

■ EXAMPLE: last does not return the last element of a list: (last '(a b)) = (b).



**(ldiff 'l\_x 'l\_y)**

- RETURNS A list of all elements in `l_x` but not in `l_y`, i.e., the list difference of `l_x` and `l_y`.
- NOTE: `l_y` must be a tail of `l_x`, i.e., **eq** to the result of applying some number of *cdr*'s to `l_x`. Note that the value of `ldiff` is always a new list structure unless `l_y` is *nil*, in which case `(ldiff l_x nil)` is `l_x` itself. If `l_y` is not a tail of `l_x`, `ldiff` generates an error.
- EXAMPLE: `(ldiff 'l_x (member 'g_foo 'l_x))` gives all elements in `l_x` up to the first `g_foo`.

### 2.1.4 List manipulation

**(rplaca 'lh\_arg1 'g\_arg2)**

- RETURNS the modified `lh_arg1`.
- SIDE EFFECT: The **car** of `lh_arg1` is set to `g_arg2`. If `lh_arg1` is a hunk then the second element of the hunk is set to `g_arg2`.

**(rplacd 'lh\_arg1 'g\_arg2)**

- RETURNS the modified `lh_arg1`.
- SIDE EFFECT: The **cdr** of `lh_arg2` is set to `g_arg2`. If `lh_arg1` is a hunk then the first element of the hunk is set to `g_arg2`.

**(attach 'g\_x 'l\_l)**

- RETURNS `l_l` whose **car** is now `g_x`, whose **cadr** is the original `(car l_l)`, and whose **cddr** is the original `(cdr l_l)`.
- NOTE: What happens is that `g_x` is added to the beginning of list `l_l` yet maintaining the same list cell at the beginning of the list.

**(delete 'g\_val 'l\_list ['x\_count])**

- RETURNS the result of splicing `g_val` from the top level of `l_list` no more than `x_count` times.
- NOTE: `x_count` defaults to a very large number, thus if `x_count` is not given, all occurrences of `g_val` are removed from the top level of `l_list`. `g_val` is compared with successive **cars** of `l_list` using the function **equal**.
- SIDE EFFECT: `l_list` is modified directly using **rplacd** when occurrences of `g_val` are spliced out. However, **delete** should be used for value and not for effect. In particular, if `l_list` is a symbol and `g_val` is the first element of the value of `l_list`, `l_list` is not modified. (See the example below.) This follows MacLISP functionality.



(delq 'g\_val 'l\_list ['x\_count])  
(dremove 'g\_val 'l\_list ['x\_count])

■ RETURNS the result of splicing g\_val from the top level of l\_list no more than x\_count times.

■ NOTE: delq (and dremove) are the same as delete except that eq is used for comparison instead of equal. The same caveats apply with respect to side effects on l\_list.

---

```
; Note that you should use the value returned by  
; delete or delq and not assume that g_val will  
; always show the deletions. For example:  
=> (setq test '(a b c a d e))  
(a b c a d e)  
=> (delete 'a test)  
(b c d e)  
; The value returned is what we would expect.  
=> test  
(a b c d e)  
; But test still has the first 'a' in the list!
```

---

(remq 'g\_x 'l\_list ['x\_count])  
(remove 'g\_x 'l\_list)

■ RETURNS A copy of l\_list with all top level elements equal to g\_x removed. remq uses eq instead of equal for comparisons.

■ NOTE: remove does not modify its arguments like delete and delq do.

(insert 'g\_object 'l\_list 'u\_comparefn 'g\_nodups)

■ RETURNS a list consisting of l\_list with g\_object destructively inserted in a place determined by the ordering function u\_comparefn.

■ NOTE: (comparefn 'g\_x 'g\_y) should return something non-nil, if g\_x can precede g\_y in sorted order; nil, if g\_y must precede g\_x. If u\_comparefn is nil, alphabetical order is used. If g\_nodups is non-nil, an element is not inserted, if an equal element is already in the list. Insert does a binary search to determine where to insert the new element.

(merge 'l\_data1 'l\_data2 'u\_comparefn)

■ RETURNS The merged list of the two input sorted lists l\_data1 and l\_data2 using binary comparison function u\_comparefn.

■ NOTE: (comparefn 'g\_x 'g\_y) should return something non-nil, if g\_x can precede g\_y in sorted order; nil, if g\_y must precede g\_x. If u\_comparefn is nil, alphabetical order is used. u\_comparefn should be thought of as "less than or equal to". merge



changes both of its data arguments.

(subst 'g\_x 'g\_y 'l\_s)  
(dsbst 'g\_x 'g\_y 'l\_s)

■ RETURNS The result of substituting g\_x for all **equal** occurrences of g\_y at all levels in l\_s.

■ NOTE: If g\_y is a symbol, **eq** is used for comparisons. The function **subst** does not modify l\_s but the function **dsbst** (destructive substitution) does.

(lsubst 'l\_x 'g\_y 'l\_s)

■ RETURNS A copy of l\_s with l\_x spliced in for every occurrence of g\_y at all levels.

Splicing in means that the parentheses surrounding the list l\_x are removed as the example below shows.

---

```
=> (subst '(a b c) 'x '(x y z (x y z) (x y z)))
((a b c) y z ((a b c) y z) ((a b c) y z))
=> (lsubst '(a b c) 'x '(x y z (x y z) (x y z)))
(a b c y z (a b c y z) (a b c y z))
```

---

(subpair 'l\_old 'l\_new 'l\_expr)

■ WHERE there are the same number of elements in l\_old as l\_new.

■ RETURNS the list l\_expr with all occurrences of an object in l\_old replaced by the corresponding one in l\_new. When a substitution is made, a copy of the value to substitute in is not made.

■ EXAMPLE: (subpair '(a c) (x y) '(a b c d)) = (x b y d).

(nconc 'l\_arg1 'l\_arg2 ['l\_arg3 ...])

■ RETURNS a list consisting of the elements of l\_arg1 followed by the elements of l\_arg2 followed by l\_arg3 and so on.

■ NOTE: The **cdr** of the last list cell of l\_argi is changed to point to l\_argi+1.

---

```
; nconc is faster than append because
; it doesn't allocate new list cells.
=> (setq lis1 '(a b c))
(a b c)
=> (setq lis2 '(d e f))
(d e f)
=> (append lis1 lis2)
(a b c d e f)
```



```
=> lis1
(a b c)
; Note that lis1 has not been changed by append.
=> (nconc lis1 lis2)
(a b c d e f)
; nconc returns the same value as append..
=> lis1
(a b c d e f)
; But in doing so alters lis1.
```

---

(reverse 'l\_arg)  
(nreverse 'l\_arg)

- RETURNS the list `l_arg` with the elements at the top level in reverse order.
- NOTE: The function `nreverse` does the reversal in place; that is, the list structure is modified.

(nreconc 'l\_arg 'g\_arg)

- EQUIVALENT TO: `(nconc (nreverse 'l_arg) 'g_arg)`.

## 2.2 Predicates

The following functions test for properties of data objects. When the result of the test is either 'false' or 'true', then `nil` is returned for 'false' and something other than `nil` (often `t`) is returned for 'true'.

(arrayp 'g\_arg)

- RETURNS `t` if `g_arg` is of type array.

(atom 'g\_arg)

- RETURNS `t` if `g_arg` is not a list or hunk object.
- NOTE: `(atom '())` returns `t`.

(bcdp 'g\_arg)

- RETURNS `t` if `g_arg` is a data object of type binary.
- NOTE: This function name is a throwback to the PDP-11 LISP system. It stands for binary code predicate.

(bigp 'g\_arg)

- RETURNS `t` if `g_arg` is a bignum.

**(dtp 'g\_arg)**

- RETURNS *t* if *g\_arg* is a list cell.
- NOTE: (dtp '()) is *nil*.

**(hunkp 'g\_arg)**

- RETURNS *t* if *g\_arg* is a hunk.

**(listp 'g\_arg)**

- RETURNS *t* if *g\_arg* is a list object or *nil*.

**(stringp 'g\_arg)**

- RETURNS *t* if *g\_arg* is a string.

**(symbolp 'g\_arg)**

- RETURNS *t* if *g\_arg* is a symbol.

**(litatom 'g\_arg)**

- RETURNS *t* if *g\_arg* is a string or symbol, not a number.

**(valuep 'g\_arg)**

- RETURNS *t* if *g\_arg* is a value cell.

**(vectorp 'v\_vector)**

- RETURNS *t* if the argument is a vector.

**(vectorip 'v\_vector)**

- RETURNS *t* if the argument is an immediate-vector.

**(keywordp 's\_sym)**

- RETURNS *t* if the argument is a symbol and is in the keyword package.

**(packagep 'k\_package)**

- RETURNS *t* if the argument is a package.

**(closurep 'c\_closure)**

- RETURNS *t* if the argument is a closure.



(fclosurep 'c\_closure)

- RETURNS *t* if the argument is an fclosure.

(hash-table-p 'H\_arg)

- RETURNS *t* if *H\_arg* is a hash table, *nil* otherwise.
- NOTE: A hash table is implemented using vectors, and the function **typep** returns vector when given a hash table.

(type 'g\_arg)

(typep 'g\_arg)

- RETURNS A symbol whose pname describes the type of *g\_arg*.

(signp s\_test 'g\_val)

- RETURNS *t* if *g\_val* is a number and the given test *s\_test* on *g\_val* returns true.
- NOTE: The fact that **signp** simply returns *nil* if *g\_val* is not a number, is probably the most important reason that **signp** is used. The permitted values for *s\_test* and what they mean are given in this table.

s_test	tested
<b>l</b>	$g\_val < 0$
<b>le</b>	$g\_val \leq 0$
<b>e</b>	$g\_val = 0$
<b>n</b>	$g\_val \neq 0$
<b>ge</b>	$g\_val \geq 0$
<b>g</b>	$g\_val > 0$

(eq 'g\_arg1 'g\_arg2)

- RETURNS *t* if *g\_arg1* and *g\_arg2* are the exact same lisp object.
- NOTE: **Eq** simply tests if *g\_arg1* and *g\_arg2* are located in exactly the same place in memory. LISP objects that print the same are not necessarily **eq**. The only objects guaranteed to be **eq** are interned symbols with the same print name. Unless a symbol is created in a special way (such as with **uconcat** or **maknam**) it is interned.

(neq 'g\_x 'g\_y)

(nequal 'g\_x 'g\_y)

- RETURNS: **neq** returns *t* if *g\_x* is not **eq** to *g\_y*, otherwise *nil*. **nequal** returns *t* if *g\_x* is not **equal** to *g\_y*, otherwise *nil*.

(equal 'g\_arg1 'g\_arg2)  
(eqstr 'g\_arg1 'g\_arg2)

- RETURNS *t* if *g\_arg1* and *g\_arg2* have the same structure as described below.
  - NOTE: *g\_arg* and *g\_arg2* are **equal** if
    - they are **eq**;
    - they are both fixnums with the same value;
    - they are both flonums with the same value;
    - they are both bignums with the same value;
    - they are both strings and are identical;
    - they are both vectors of identical size with **equal** elements;
    - they are both immediate vectors of identical size with identical elements;
    - they are both arrays and **eq**;
    - they are both hunks and **eq**; or
    - they are both lists and their **cars** and **cdrs** are **equal**.
- If a data type is not enumerated above, they must be **eq** to be **equal**.

(eq1 'g\_arg1 'g\_arg2)

- RETURNS *t* if *g\_arg1* and *g\_arg2* are **eq** or if they are numbers of the same type with the same value.

---

```

; eq is much faster than equal, especially
; in compiled code. However, you cannot use
; eq to test for equality of numbers outside
; of the range -1024 to 1023. equal always
; works.
; eq1 is also much more efficient than equal, and it can test for
; equality of numbers.
=> (eq 1023 1023)
t
=> (eq 1024 1024)
nil
=> (equal 1024 1024)
t
=> (eq1 1024 1024)
t

```

---

(not 'g\_arg)  
(null 'g\_arg)

- RETURNS *t* if *g\_arg* is *nil*.



(member 'g\_arg1 'l\_arg2)  
(memq 'g\_arg1 'l\_arg2)

- RETURNS that part of the `l_arg2` beginning with the first occurrence of `g_arg1`. If `g_arg1` is not in the top level of `l_arg2`, `nil` is returned.
- NOTE: **member** tests for equality with **equal**; **memq** tests for equality with **eq**.

## 2.3 Symbols and strings

In many of the following functions, the distinction between symbols and strings is somewhat blurred. For FRANZ LISP, a string is a null terminated sequence of characters, stored as compactly as possible. Strings are used as constants in FRANZ LISP. They **eval** to themselves. A symbol has additional structure: a value, property list, function binding, package-cell, as well as its external representation (or print name). If a symbol is given to one of the string manipulation functions below, its print name is used as the string.

Another popular way to represent strings in LISP is as a list of fixnums which represent characters. The suffix *n* to a string manipulation function indicates that it returns a string in this form.

### 2.3.1 Symbol and string creation

(concat ['stn\_arg1 ... ])  
(uconcat ['stn\_arg1 ... ])  
(strcat ['stn\_arg1 ... ])

- RETURNS A symbol whose print name (or in the case of **strcat**, a string) is the result of concatenating the print names, string characters, or numerical representations of the `stn_argi`.
- NOTE: If no arguments are given, a symbol with a null pname is returned. **concat** interns (see **Intern**) the symbol in the current package; the function **uconcat** does the same thing but does not intern the symbol.
- EXAMPLE: (concat 'abc (add 3 4) "def") = abc7def.

(concatl 'l\_arg)

- EQUIVALENT TO: (apply 'concat 'l\_arg).

(implode 'l\_arg)  
(implodes 'l\_arg)  
(maknam 'l\_arg)

- WHERE `l_arg` is a list of symbols, strings or small fixnums.
- RETURNS the symbol whose print name (or in the case of **implodes**, the string) that is the result of concatenating the first characters of the print names of the symbols and strings in the list. Any fixnums are converted to the equivalent ASCII character. In order to concatenate entire strings or print names, use the function **concat**.



- NOTE: **implode** interns the symbol it creates, **maknam** does not.

(copysymbol 's\_arg 'g\_pred)  
(copy-symbol 's\_arg 'g\_pred)

- RETURNS an uninterned symbol with the same print name as **s\_arg**. If **g\_pred** is non *nil*, then the value, function binding, and property list of the new symbol are made **eq** to those of **s\_arg**.
- NOTE: **copy-symbol** is the Common LISP name for this function.

(ascii 'x\_charnum)

- WHERE **x\_charnum** is between 0 and 255.
- RETURNS a symbol whose print name is the single character whose fixnum representation is **x\_charnum**.

(intern 's\_arg ['k\_package])

- RETURNS multiple-values of which the first is **s\_arg** (see Chapter 17).
- SIDE EFFECT: **s\_arg** is installed in the package **k\_package** or (by default) the current package.
- NOTE: When a symbol is interned, a mapping between the external print name and the symbol itself is established. Most symbols are interned so that when the user types in say, **foo**, this is the same symbol as previously used. It is possible to change an existing **foo** to be uninterned; in that case when a token **foo**, is read in, a new symbol, unrelated to the old one except by its coincidentally matching print name, is created. The two symbols will ordinarily have different values, etc. One can refer to the old **foo** only by having an internal pointer to it. As far as the reader is concerned, there is no **foo** like the old **foo**.

(remob 's\_symbol)

- RETURNS **s\_symbol**.
- SIDE EFFECT: **s\_symbol** is removed from the current package. Historically, the name **remob** comes from "remove from the object list."

(rematom 's\_arg)

- RETURNS **t** if **s\_arg** is indeed an atom.
- SIDE EFFECT: **s\_arg** is put on the free atom list, effectively reclaiming an atom cell.
- NOTE: This function does not check to see if **s\_arg** is accessible. While **rematom** enables you to reclaim a small amount of storage, and can be used effectively with **gensymed** atoms, you must be extremely cautious. If you use it on an interned atom which is referenced by some s-expression, you may find that one or more different atoms are synonymous. This can lead to errors which are very difficult to detect. This function should be used only when storage optimization is important and you are



creating many atoms with short lifetimes.

The following functions are recommended for use in the orderly management of generated symbols:

**(gensym [*s\_leader*])**

- RETURNS a new uninterned atom beginning with the first character of *s\_leader*'s pname, or beginning with *g* if *s\_leader* is not given.
- NOTE: The symbol generated ends with a six-digit suffix which is the number of times **gensym** has been called.

**(gentemp [*s\_name* [*k\_package*]])**

- RETURNS a new symbol whose name will be *s\_name* (or "T" if *s\_name* is not given) with an integer concatenated to the right. This symbol is interned in the specified package (or *\*package\** if *k\_package* is not given).
- NOTE: This function guarantees that the symbol created will be a new symbol not already existent in the package. It uses a counter like **gensym**, but the counter is incremented repeatedly if a symbol is generated that already exists. There is no way to reset the counter used by **gentemp**. The prefix *s\_name* is not remembered from one call to the next and defaults to "T" if not given.

### 2.3.1.1 Symbol generating facility

The functions described below are an alternative to the **gensym** facility. They generate new symbols by attaching counter numbers as suffixes to the symbols' names. An example follows of how the functions are used.

---

```
;      initialize new symbol counters
=> (initsym 'joe '(john 5))
(joe0 john5)
;      create a new symbol
=> (newsym 'john)
john6
;      symbol need not be initsym'd
=> (newsym 'chuck)
chuck0
;      get current symbol
=> (oldsym 'john)
john6
;      get all symbols between 0 and counter
=> (allsym 'john)
(john0 john1 john2 john3 john4 john5 john6)
;      get all symbols between 5 and counter
=> (allsym '(john 5))
(john5 john6)
```



```

;      remob all interned symbols associated with
;      joe and from john4 to the current john symbol.
;      Returns symbols with symbol counters before doing remsym.
=> (remsym 'joe '(john 4))
( #:joe0 #:john6)
=> (symstat 'joe 'john)
((joe nil) (john 3))

```

---

**(initsym 'ls\_arg1 ...)**

- WHERE each ls\_arg is a list (n\_counter s\_arg) or a symbol s\_arg (which is equivalent to (0 s\_arg)).
- RETURNS a list of interned symbols concatenated to its individual symbol counter.

```

=> (initsym 'joe '(john 5))
(joe0 john5)

```

---

- NOTE: See also **newsym**, **oldsym**, **allsym**, **remsym**, and **symstat** functions.

**(newsym 's\_arg)**

- RETURNS an interned symbol formed by concatenating the name s\_arg to its symbol counter. The symbol counter is stored on the property list of s\_arg under symctr. If there is no counter, a count of 0 is used and added to the property list. Thus, a symbol need not be initsym'ed.

```

=> (initsym 'joe '(john 5))
(joe0 john5)
=> (newsym 'john)
john6
=> (newsym 'joe)
joel

```

---

- NOTE: See also **initsym**, **oldsym**, **allsym**, **remsym**, and **symstat** functions.

**(oldsym 's\_arg)**

- RETURNS the symbol using the current symbol counter for s\_arg, rather than creating a new symbol. If no symbol counter exists for s\_arg, then s\_arg is returned.
- NOTE: See also **initsym**, **newsym**, **allsym**, **remsym**, and **symstat** functions.



(allsym 'ls\_arg)

- WHERE `ls_arg` is a list (`s_arg n_counter`) or a symbol `s_arg` (equivalent to (`s_arg 0`)).
- RETURNS a list of all the created identifiers between `n_counter` and the current symbol counter for `s_arg`.
- NOTE: See also `initsym`, `newsym`, `oldsym`, `remsym`, and `symstat` functions.

(remsym 'ls\_arg ...)

- WHERE each `ls_arg` is a list (`s_arg n_counter`) or a symbol `s_arg` (equivalent to (`s_arg 0`)).
- RETURNS a corresponding list of symbols `s_arg` with the current symbol counters.
- SIDE EFFECT: This function uninterns all the created identifiers between 0 and the current symbol counter for `s_arg`.
- NOTE: See also `initsym`, `newsym`, `oldsym`, `allsym`, and `symstat` functions.

(symstat 's\_arg ...)

- RETURNS a list of pairs for each `s_arg` consisting of (`s_arg symctr`) where `symctr` is `s_arg`'s current symbol counter.
- NOTE: See also `initsym`, `newsym`, `oldsym`, `allsym`, and `remsym` functions.

### 2.3.2 String and symbol predicates

(boundp 's\_name)

- RETURNS `nil` if `s_name` is unbound; that is, it has never been given a value. If `s_name` has the value `g_val`, then (`nil . g_val`) is returned. See also `makunbound`.

(alphalessp 'st\_arg1 'st\_arg2)

- RETURNS `t` if the 'name' of `st_arg1` is alphabetically less than the name of `st_arg2`. If `st_argi` is a symbol, then its 'name' is its print name. If `st_argi` is a string, then its 'name' is the string itself.
- NOTE: This function has been supplanted by `string<`.

(string= 't\_string1 't\_string2)

(string/= 't\_string1 't\_string2)

(string< 't\_string1 't\_string2)

(string> 't\_string1 't\_string2)

(string<= 't\_string1 't\_string2)

(string>= 't\_string1 't\_string2)

- RETURNS `nil` unless `t_string1` is respectively lexicographically equal, not equal, less than, greater than, not greater than, or not less than `t_string2`. If the respective condition is satisfied, the function returns an index to the first character position in



the strings that failed to match.

■ NOTE: An error is signalled if the arguments are not strings.

**(string-length 't\_string)**

■ RETURNS the length of string **t\_string**.

■ NOTE: An error is signalled if the argument is not a string. The terminating ASCII NUL byte of the string is not counted in the length of the string.

**(string-upcase 't\_string)**

**(string-downcase 't\_string)**

■ RETURNS a string just like **t\_string** with all lowercase (or uppercase for **string-downcase**) characters replaced by the corresponding uppercase (or lowercase) characters.

**(string-capitalize 't\_string)**

■ RETURNS a copy of **t\_string** that has every initial letter of every word in upper case and all other letters of every word in lower case.

■ NOTE: A word is defined to be a consecutive sequence of alphanumeric characters or digits delimited by non-alphanumeric characters or the beginning or end of the string. Case is only meaningful for alphabetic characters—non-alphabetic characters are not affected.

**(nstring-upcase 't\_string)**

**(nstring-downcase 't\_string)**

**(nstring-capitalize 't\_string)**

■ NOTE: These functions are identical to **string-upcase**, **string-downcase**, and **string-capitalize**, respectively, except that they destructively modify, and return, the argument **t\_string**.

### 2.3.3 Symbol and string accessing

**(symeval 's\_arg)**

**(symbol-value 's\_arg)**

■ RETURNS the value of symbol **s\_arg**. **symbol-value** is the Common LISP name for this function.

■ NOTE: It is illegal to ask for the value of an unbound symbol. This function has the same effect as **eval**, but compiles into much more efficient code.



(get\_pname 's\_arg)  
(symbol-name 's\_arg)

- RETURNS the string that is the print name of `s_arg`.
- NOTE: **symbol-name** is the Common LISP name for this function.

(getd 's\_arg)  
(symbol-function 's\_arg)

- RETURNS the function definition of `s_arg`. If there is no function definition, **getd** returns *nil*; **symbol-function**, the Common LISP function signals an error in this case.
- NOTE: The function definition may turn out to be an array header. You might wish to use **fboundp** to see if a function definition exists in a Common LISP world, rather than **getd**.

(symbol-package 's\_name)

- RETURNS the contents of the package cell for the symbol `s_name`. This will be *nil* if there is no package associated with `s_name`.

(fboundp 's\_arg)

- RETURNS *t* if `s_arg` is bound to a function.
- NOTE: If the function slot in the symbol `s_arg` contains an array object, this function will return *nil*.

(getchar 's\_arg 'x\_index)  
(nthchar 's\_arg 'x\_index)  
(getcharn 's\_arg 'x\_index)

- RETURNS The `x_index`th character of the print name of `s_arg` or *nil* if `x_index` is less than 1 or greater than the length of `s_arg`'s print name.
- NOTE: **getchar** and **nthchar** return a symbol with a single character print name; **getcharn** returns the fixnum representation of the character.

(substring 'st\_string 'x\_index ['x\_length])  
(substringn 'st\_string 'x\_index ['x\_length])

- RETURNS a string of length at most `x_length` starting at `x_index`th character in the string.
- NOTE: If `x_length` is not given, all of the characters for `x_index` to the end of the string are returned. If `x_index` is negative, the string begins at the `x_index`th character from the end. If `x_index` is out of bounds, *nil* is returned.
- NOTE: **substring** returns a list of symbols; **substringn** returns a list of fixnums. If **substringn** is given a 0 `x_length` argument, then a single fixnum, which is the `x_index`th character, is returned.



(char-index 't\_string 'stx\_char)  
(char-rindex 't\_string 'stx\_char)

- RETURNS the index of stx\_char in t\_string, from the beginning of the string, in the case of char-index, and the end of the string in the case of char-rindex. If stx\_char is a fixnum it will be treated as the ASCII code for the character. If stx\_char is a symbol or string, the first character of the print name or the string is used.

(substringp 't\_string1 't\_string2)

- RETURNS *t* if t\_string1 is contained in t\_string2, *nil* otherwise.

(string 'st\_symbol-or-string)

- RETURNS a string, given a string or symbol. In case of a symbol argument, it returns the symbol's print name.

### 2.3.4 Symbol and string manipulation

(set 's\_arg1 'g\_arg2)

- RETURNS g\_arg2.
- SIDE EFFECT: The value of s\_arg1 is set to g\_arg2.

(setq s\_atm1 'g\_val1 [ s\_atm2 'g\_val2 ... ... ])

- WHERE the arguments are pairs of atom names and expressions.
- RETURNS the last g\_vali.
- SIDE EFFECT: Each s\_atmi is set to have the value g\_vali.
- NOTE: **set** evaluates all of its arguments; **setq** does not evaluate the s\_atmi.
- NOTE: The assignments are done sequentially from left to right.

(desetq sl\_pattern1 'g\_exp1 [... ...])

- RETURNS g\_expn
- SIDE EFFECT: This acts just like **setq** if all the sl\_patterni are symbols. If sl\_patterni is a list, then it is a template which should have the same structure as g\_expi. The symbols in sl\_patterni are assigned to the corresponding parts of g\_expi. (See also **setf**.)
- EXAMPLE: (desetq (a b (c . d)) '(1 2 (3 4 5))) sets a to 1, b to 2, c to 3, and d to (4 5).

(setplist 's\_atm 'l\_plist)

- RETURNS l\_plist.
- SIDE EFFECT: The property list of s\_atm is set to l\_plist.



(makunbound 's\_arg)

- RETURNS `s_arg`
- SIDE EFFECT: The value of `s_arg` is made *unbound*. If the interpreter attempts to evaluate `s_arg` before it is again given a value, an unbound variable error occurs.

(explode 'g\_arg)

(explodec 'g\_arg)

(exploden 'g\_arg)

(escape-exploden 'g\_arg)

(qualify-explode 'g\_arg)

(qualify-explodec 'g\_arg)

(qualify-exploden 'g\_arg)

(qualify-escape-exploden 'g\_arg)

- RETURNS a list of the characters used to print out `g_arg`.
- NOTE: If the function name ends in an *n*, then the function returns a fixnum representation of the character instead of the character itself. Also, the functions beginning with "qualify" return a list of characters or fixnums which include package qualifiers.
- The functions **explode**, **escape-exploden**, **qualify-explode** and **qualify-escape-exploden** return a list of characters or fixnums that **print** would use to print the argument, which include all necessary escape characters. The functions **explodec**, **exploden**, **qualify-explodec** and **qualify-exploden** return a list of characters or fixnums that **patom** would use to print the argument (that is, no escape characters).

---

```
=> (setq x '|quote this \| ok?|)
|quote this \| ok?|
=> (explode x)
(q u o t e | \| | t h i s | \| | | \| \| \| \| \| | | o k ?)
; Note that | \| just means the single
; character 'backslash'; | \| just means
; the single character 'vertical bar',
; and | | means the single character 'space'.
=> (explodec x)
(q u o t e | | t h i s | | | \| | | o k ?)
=> (exploden x)
(113 117 111 116 101 32 116 104 105 115 32 124 32 111 107 63)
```

---



```
(aexplode 's_arg)
(aexplodec 's_arg)
(aexploden 's_arg)
(escape-aexploden 's_arg)
(qualify-aexplode 's_arg)
(qualify-aexplodec 's_arg)
(qualify-aexploden 's_arg)
(qualify-escape-aexploden 's_arg)
(qualify-always-escape-aexploden 's_arg)
```

- RETURNS a list of the characters used to print out `s_arg`.
- NOTE: These functions are identical to the preceding set of functions except that they take only symbols as arguments.

## 2.4 Vectors

See Chapter 9 for a discussion of vectors.

### 2.4.1 Vector creation

```
(new-vector 'x_size ['g_fill ['g_prop]])
```

- RETURNS A **vector** of length `x_size`. Each data entry is initialized to `g_fill`, or to *nil*, if the argument `g_fill` is not present. The vector's property is set to `g_prop`, or to *nil*, by default.

```
(new-vectori-byte 'x_size ['g_fill ['g_prop]])
(new-vectori-word 'x_size ['g_fill ['g_prop]])
(new-vectori-long 'x_size ['g_fill ['g_prop]])
(new-vectori-float 'x_size ['g_fill ['g_prop]])
(new-vectori-double 'x_size ['g_fill ['g_prop]])
```

- RETURNS a **vectori** with `x_size` elements in it. The actual memory requirement is two long words + `x_size` • (*n* bytes), where *n* is 1 for **new-vector-byte**, 2 for **new-vector-word**, 4 for **new-vectori-long** or **new-vectori-float**, or 8 for **new-vectori-double**. Each data entry is initialized to `g_fill`, or to zero, if the argument `g_fill` is not present. The vector's property is set to `g_prop`, or *nil*, by default.

- NOTE: **new-vectori-float** and **new-vectori-double** are intended to be used in passing float and double arrays to C routines.

```
(vector ['g_val0 'g_val1 ...])
```

- RETURNS a **vector** with as many data elements as there are arguments. It is quite possible to have a vector with no data elements. The vector's property is a null list.



(**vectori-byte** ['x\_val0 'x\_val1 ...])  
(**vectori-word** ['x\_val0 'x\_val1 ...])  
(**vectori-long** ['x\_val0 'x\_val1 ...])  
(**vectori-float** ['f\_val0 'f\_val1 ...])  
(**vectori-double** ['f\_val0 'f\_val1 ...])

■ RETURNS a **vectori** with as many data elements as there are arguments. The arguments are required to be fixnums, except in the case of float and double, where they must be flonums. Only the low order byte or word is used in the case of **vectori-byte** and **vectori-word**. The vector's property is null.

### 2.4.2 Vector predicate

(**vectorp** 'g\_arg)

■ RETURNS *t* if the argument is a vector, *nil* otherwise.

### 2.4.3 Vector reference

(**vref** 'v\_vect 'x\_index)  
(**vrefi-byte** 'V\_vect 'x\_bindex)  
(**vrefi-word** 'V\_vect 'x\_windex)  
(**vrefi-long** 'V\_vect 'x\_lindex)  
(**vrefi-float** 'V\_vect 'x\_lindex)  
(**vrefi-double** 'V\_vect 'x\_lindex)

■ RETURNS the desired data element from a vector. The indices must be fixnums. Indexing is zero-based. The **vrefi** functions sign-extend the data.

(**vprop** 'Vv\_vect)

■ RETURNS the LISP property associated with a vector.

(**vget** 'Vv\_vect 'g\_ind)

■ RETURNS the value stored under *g\_ind* if the LISP property associated with *Vv\_vect* is a disembodied property list.

(**vsize** 'Vv\_vect)

■ RETURNS the number of data elements in the vector.

■ NOTE: If given an immediate vector as argument, the number of pointer-sized elements in the vector is returned. The size of a pointer will vary from machine to machine (on VAXen and most MC680x0 it is 4 bytes). The size of immediate vectors is best obtained by the set of functions below.



(**vsized-byte** 'V\_vect)  
 (**vsized-word** 'V\_vect)  
 (**vsized-float** 'V\_vect)  
 (**vsized-double** 'V\_vect)

■ RETURNS the number of data elements in the immediate vector. These functions return the number of data elements of the corresponding type in the immediate vector. For example, **vsized-word** returns the number of words in the immediate vector. Fractional values are truncated so that an immediate vector containing nine bytes will appear to have four words when **vsized-word** is applied to it.

#### 2.4.4 Vector modification

(**vset** 'v\_vect 'x\_index 'g\_val)  
 (**vseti-byte** 'V\_vect 'x\_bindex 'x\_val)  
 (**vseti-word** 'V\_vect 'x\_windex 'x\_val)  
 (**vseti-long** 'V\_vect 'x\_lindex 'x\_val)  
 (**vseti-float** 'V\_vect 'x\_findex 'f\_val)  
 (**vseti-double** 'V\_vect 'x\_dindex 'f\_val)

■ RETURNS the new value of the indexed element.  
 ■ SIDE EFFECT: The indexed element of the vector is set to the value. The immediate vector functions interpret the index in the units of the corresponding data type, e.g. **vseti-long** interprets the index as a longword offset into the immediate vector. The immediate vector functions **vseti-byte** and **vseti-word** store the low order byte or word of x\_val into the indexed data element of the immediate vector.

(**vsetprop** 'Vv\_vect 'g\_value)

■ RETURNS g\_value. This should be either a symbol or a disembodied property list whose **car** is a symbol identifying the type of the vector.  
 ■ SIDE EFFECT: The property list of Vv\_vect is set to g\_value.

(**vputprop** 'Vv\_vect 'g\_value 'g\_ind)

■ RETURNS g\_value.  
 ■ SIDE EFFECT: If the vector property of Vv\_vect is a disembodied property list, then vputprop adds the value g\_value under the indicator g\_ind. Otherwise, the old vector property is made the first element of the list.

#### 2.4.5 Miscellaneous

(**vectori-byte-to-string** 'Vv\_vect)

■ RETURNS a string consisting of the bytes matching those in Vv\_vect up to the first null byte.



## 2.5 Hash tables

A hash table is an object that can efficiently map one object to another. Each hash table is a collection of entries, each of which associates a unique *key* with a *value*. There are functions to add, delete, and find entries based on a particular key. Finding a value in a hash table is relatively fast compared to looking up values in, for example, an assoc list or property list.

The hash table is **not** a true data type, but rather a type which is constructed from objects of the type vector. Because of this, the vector predicate returns a non-*nil* value when given a hash table. It should be noted that using **vset** to set a element of the hash table will yield unpredictable results.

Adding a key to a hash table modifies the hash table, and is therefore a destructive operation.

There are two different kinds of hash tables: those that use the function **equal** for the comparing of keys, and those that use **eq**, the default. When a hash table is created, the type of comparator is set. If **eq** is chosen as the comparator, and a lookup of a key is being performed, then the given key is compared to the keys in the table using **eq**.

Hashing provides an efficient basis for the construction of the **packages** facility and for various sorts of data retrieval techniques.

This hash table package is completely compatible with the one in Common LISP.

### 2.5.1 Hash table functions

(**make-hash-table** [keyword value ... ...])

■ RETURNS a hash table object.

**:size** x\_size

■ The x\_size argument sets the initial size of the hash table. Specifically, this is a suggestion for the number of buckets you would like the hash table to have, and the actual number will be *close* to this.

**:test** u\_compare

■ The u\_compare argument determines how the keys in the hash table are compared, and must be either **eq** or **equal**, defaulting to **eq**.

**:rehash-size** x\_rehash\_size

■ The x\_rehash\_size argument specifies how much the size of the hash table is increased when overflow occurs. If it is an integer greater than zero then it is the number of entries to grow the hash table. If it is a floating point number greater than one then it is taken to be the ratio of the new size to the old size.



**:rehash-threshold** *x\_rehash\_threshold*

■ The *x\_rehash\_threshold* argument specifies how full the hash table can get before it must grow. The process of growing a hash table is called *rehashing*. It can be an integer greater than zero and less than the **:rehash-size** argument (in which case it will be scaled whenever the hash table is grown), or it can be a floating point number between zero and one.

■ NOTE: For an explanation of keyword arguments, see §8.2.

**(gethash 'g\_key 'H\_htab [ 'g\_defval ])**

■ RETURNS two values: first, the value associated with the key *g\_key* in hash table *H\_htab*, or *nil* if the key was not in the table, and then a predicate value to indicate whether or not the search was successful. If *g\_defval* is given and there is no entry in the hash table, then *g\_defval* is returned. As is the standard with a multiple-value return, if only one value is expected, the first is used.

■ NOTE: **setf** may be used to set the value associated with a key.

**(addhash 'g\_key 'H\_htab 'g\_val)**

■ RETURNS *g\_key*, after installing it with its value *g\_val* to the hash table.

■ EQUIVALENT TO: (**setf** (**gethash** 'g\_key 'H\_htab) 'g\_val).

**(remhash 'g\_key 'H\_htab)**

■ RETURNS *t* if there was an entry for *g\_key* in the hash table *H\_htab*, *nil* otherwise. In the case of a match, the entry and associated object are removed from the hash table.

**(maphash 'u\_fun 'H\_htab [ 'g\_rest ])**

■ RETURNS *nil*.

■ NOTE: The function *u\_fun* is applied to every element in the hash table *H\_htab*. The function should expect two arguments: the key and value of an element. The mapped function should not add or delete objects from the table because the results would be unpredictable. It is, however, acceptable to use **remhash** or **addhash** on the entry currently being mapped over.

□ The optional third argument is a mechanism for passing arguments to the mapping function.

**(clrhash 'H\_htab)**

■ RETURNS the hash table cleared of all entries.



(hash-table-count 'H\_htab)

■ RETURNS the number of entries in H\_htab. Given a hash table with no entries, this function returns zero.

---

```

; Make a vanilla hash table using eq to compare items.
=> (setq black-box (make-hash-table :size 20))
#<eq hash-table with 0 entries and 20 buckets @ #xe45cc>
=> (hash-table-p black-box)
t
=> (hash-table-count black-box)
0
=> (setf (gethash 'anykey black-box) '(this list is the value))
anykey
=> (gethash 'anykey black-box)
(this list is the value)
=> (hash-table-count black-box)
1
=> (addhash 'composer black-box 'franz)
composer
=> (gethash 'composer black-box)
franz
=> (maphash '(lambda (key val) (msg "key=" key ",value=" val N))
black-box)
key=composer,value=franz
key=anykey,value=(this list is the value)
nil
=> (clrhash black-box)
#<eq hash-table with 0 entries and 20 buckets @ #xe45cc>
=> (hash-table-count black-box)
0
=> (maphash '(lambda (key val) (msg "key=" key ",value=" val N))
black-box)
nil
; Here is an example using equal as the comparator.
=> (setq ht (make-hash-table :size 10 :test #'equal))
#<eq hash-table with 0 entries and 10 buckets @ #xf69ec>
=> (setf (gethash '(this is a key) ht) '(and this is the value))
(this is a key)
=> (gethash '(this is a key) ht)
(and this is the value)
; The reader makes a new list each time you type it.
=> (setq x '(this is a key))
(this is a key)
=> (setq y '(this is a key))
(this is a key)
; These two lists are really different lists:
; they are equal but not eq.
=> (equal x y)
t
=> (eq x y)

```



```

nil
; Since we are using equal to compare keys, we are okay.
=> (gethash x ht)
    (and this is the value)
=> (gethash y ht)
    (and this is the value)

```

---

(sxhash 'g\_arg)

■ RETURNS a fixnum representing the hash code for `g_arg`. The hash code is computed in such a way that if two objects are **equal**, they will have the same hash code. In other words, `(equal val1 val2)` implies `(= (sxhash val1) (sxhash val2))`.

## 2.6 Arrays

See Chapter 9 for a complete description of arrays. Some of these functions are part of a MacLisp array compatibility package representing only one simple way of using the array structure of FRANZ LISP.

### 2.6.1 Array creation

(marray 'g\_data 's\_access 'g\_aux 'x\_length 'x\_delta)

■ RETURNS an array type with the fields set up from the above arguments whose meanings are indicated in chapter 9 (see also §1.1.11).

(\*array 's\_name 's\_type 'x\_dim1 ... 'x\_dimn)  
(array s\_name s\_type x\_dim1 ... x\_dimn')

■ WHERE `s_type` may be one of `t`, `nil`, `fixnum`, `flonum`, `fixnum-block`, or `flonum-block`.

■ RETURNS an array of type `s_type` with `n` dimensions of extents given by the `x_dimi`.

■ SIDE EFFECT: If `s_name` is non `nil`, the function definition of `s_name` is set to the array structure returned.

■ NOTE: These functions create a MacLISP compatible array. In FRANZ LISP arrays of type `t`, `nil`, `fixnum`, and `flonum` are equivalent and the elements of these arrays can be any type of lisp object. `Fixnum-block` and `flonum-block` arrays are restricted to `fixnums` and `flonums` respectively and are used mainly to communicate with foreign functions (see §18.4.3).

■ NOTE: `*array` evaluates its arguments, `array` does not.



## 2.6.2 Array predicate

(arrayp 'g\_arg)

- RETURNS *t* if the argument is an array, *nil* otherwise. Note that a vector is not an array in FRANZ LISP.

## 2.6.3 Array accessors

(getaccess 'a\_array)

(getaux 'a\_array)

(getdelta 'a\_array)

(getdata 'a\_array)

(getlength 'a\_array)

- RETURNS the field of the array object *a\_array* given by the function name.

(arrayref 'a\_name 'x\_ind)

- RETURNS the *x\_ind*th element of the array object *a\_name*. *x\_ind* of zero accesses the first element.
- NOTE: **arrayref** uses the data, length, and delta fields of *a\_name* to determine which object to return.

(arraycall s\_type 'as\_array 'x\_ind1 ... )

- RETURNS the element selected by the indices from the array *a\_array* of type *s\_type*.
- NOTE: If *as\_array* is a symbol, then the function binding of this symbol should contain an array object.
- *s\_type* is ignored by **arraycall** but is included for compatibility with MacLISP.

(arraydims 'a\_name)

- RETURNS a list of the type and bounds of the array *a\_name*.

(listarray 'sa\_array ['x\_elements])

- RETURNS a list of all of the elements in array *sa\_array*. If *x\_elements* is given, then only the first *x\_elements* are returned.

---

```
; This creates a 3 by 4 array of general lisp objects.  
=> (array ernie t 3 4)  
array[12]
```

```
; The array header is stored in the function definition  
; slot of the symbol ernie.
```



```

=> (arrayp (getd 'ernie))
t
=> (arraydims (getd 'ernie))
(t 3 4)

; Store in ernie[2][2] the list (test list).
=> (store (ernie 2 2) '(test list))
(test list)

; Check to see if it is there.
=> (ernie 2 2)
(test list)

; Now use the low-level function arrayref
; to find the same element.
; Arrays are zer-based and row-major (the last
; subscript varies the fastest); thus
; element [2][2] is the 10th element, starting at 0.
=> (arrayref (getd 'ernie) 10)
(ptr to) (test list)
; The result is a value cell (thus the (ptr to)).

```

---

## 2.6.4 Array manipulation

```

(putaccess 'a_array 'su_func)
(putaux 'a_array 'g_aux)
(putdata 'a_array 'g_arg)
(putdelta 'a_array 'x_delta)
(putlength 'a_array 'x_length)

```

- RETURNS the second argument to the function.
- SIDE EFFECT: The field of the array object given by the function name is replaced by the second argument to the function.

```
(store 'l_arexp 'g_val)
```

- WHERE l\_arexp is an expression that references an array element.
- RETURNS g\_val.
- SIDE EFFECT: The array location that contains the element that l\_arexp references is changed to contain g\_val.

```
(fillarray 's_array 'l_itms)
```

- RETURNS s\_array.
- SIDE EFFECT: The array s\_array is filled with elements from l\_itms. If there are not enough elements in l\_itms to fill the entire array, then the last element of l\_itms is used to fill the remaining parts of the array.



## 2.7 Hunks

Hunks are vector-like objects whose size can range from 1 to 128 elements. Internally, hunks are allocated in sizes that are powers of 2. In order to create hunks of a given size, a hunk with at least that many elements is allocated, and a distinguished symbol *EMPTY* is placed in those elements not requested. Most hunk functions respect those distinguished symbols, but there are two **\*makhunk** and **\*rplacx** that overwrite the distinguished symbol.

### 2.7.1 Hunk creation

(hunk 'g\_val1 ['g\_val2 ... 'g\_valn])

- RETURNS a hunk of length *n* whose elements are initialized to the *g\_vali*.
- NOTE: The maximum size of a hunk is 128.
- EXAMPLE: (hunk 4 'sharp 'keys) = {4 sharp keys}.

(makhunk 'xl\_arg)

- RETURNS a hunk of length *xl\_arg* initialized to all *nils* if *xl\_arg* is a fixnum. If *xl\_arg* is a list, then a hunk of size (length 'xl\_arg) is returned, initialized to the elements in *xl\_arg*.
- NOTE: (makhunk '(a b c)) is equivalent to (hunk 'a 'b 'c).
- EXAMPLE: (makhunk 4) = {nil nil nil nil}.

(\*makhunk 'x\_arg)

- RETURNS a hunk of size  $2^{x\_arg}$  initialized to *EMPTY*.
- NOTE: This is only to be used by such functions as **hunk** and **makhunk**, which create and initialize hunks for users.

### 2.7.2 Hunk accessor

(cxr 'x\_ind 'h\_hunk)

- RETURNS element *x\_ind* (starting at 0) of hunk *h\_hunk*.

(hunk-to-list 'h\_hunk)

- RETURNS a list consisting of the elements of *h\_hunk*.

### 2.7.3 Hunk manipulators

(rplacx 'x\_ind 'h\_hunk 'g\_val)  
 (\*rplacx 'x\_ind 'h\_hunk 'g\_val)

- RETURNS h\_hunk.
- SIDE EFFECT: Element x\_ind (starting at 0) of h\_hunk is set to g\_val.
- NOTE: rplacx does not modify one of the distinguished (EMPTY) elements whereas \*rplacx does.

(hunksize 'h\_arg)

- RETURNS the size of the hunk h\_arg.
- EXAMPLE: (hunksize (hunk 1 2 3)) = 3

## 2.8 Bcds

A bcd object contains a pointer to compiled code and to the type of function object the compiled code represents.

(getdisc 'y\_bcd)  
 (getentry 'y\_bcd)

- RETURNS the field of the bcd object given by the function name.

(putdisc 'y\_func 's\_discipline)

- RETURNS s\_discipline.
- SIDE EFFECT: Sets the discipline field of y\_func to s\_discipline.

## 2.9 Structures

There are three common structures constructed out of list cells: the assoc list, the property list, and the tconc list. The functions below manipulate these structures.

### 2.9.1 Association lists

An association list (or alist) is a Common LISP data structure. It has the form

((key1 . value1) (key2 . value2) (key3 . value3) ... (keyn . valuen)).

(assoc 'g\_arg1 'l\_arg2)  
 (assq 'g\_arg1 'l\_arg2)  
 (rassq 'g\_arg1 'l\_arg2)

- RETURNS the first top level element of l\_arg2 whose car is equal (with assoc) or eq (with assq) to g\_arg1.



■ NOTE: Usually `l_arg2` has an a-list structure and `g_arg1` acts as key. `rassq` is similar to `assq`, with the *r* indicating 'reversal', in that it looks at `cdr`'s instead of `car`'s in `l_arg2`.

`(sassoc 'g_arg1 'l_arg2 'sl_func)`

- RETURNS the result of `(cond ((assoc 'g_arg 'l_arg2) (apply 'sl_func nil)))`.
- NOTE: `sassoc` is written as a macro.

`(sassq 'g_arg1 'l_arg2 'sl_func)`

- RETURNS the result of `(cond ((assq 'g_arg 'l_arg2) (apply 'sl_func nil)))`
- NOTE: `sassq` is written as a macro.

---

```

; assoc or assq is given a key and an assoc list and returns
; the key and value item if it exists.
; They differ only in how they test
; for equality of the keys.
=> (setq alist '((alpha . a) ((complex key) . b) (junk . x)))
((alpha . a) ((complex key) . b) (junk . x))

; You should use assq when the key is an atom.
=> (assq 'alpha alist)
(alpha . a)

; But it may not work when the key is a list.
=> (assq '(complex key) alist)
nil

; However, assoc always works.
=> (assoc '(complex key) alist)
((complex key) . b)

```

---

`(sublis 'l_alst 'l_exp)`

- WHERE `l_alst` is an a-list.
- RETURNS the list `l_exp`, `((g_key1 . g_value2) ...)` with every occurrence of `keyi` replaced by `valuei`.
- NOTE: A new list structure is returned to prevent modification of `l_exp`. When a substitution is made, a copy of the value to substitute in, is not made.

## 2.9.2 Property list

A property list consists of an alternating sequence of keys and values. Normally a property list is stored on a symbol. A list is a 'disembodied' property list if it contains an odd number of elements, the first of which is ignored.



(plist 's\_name)

(symbol-plist 's\_name)

- RETURNS the property list of s\_name.
- NOTE: **symbol-plist** is the Common LISP name for this function.

(setplist 's\_atm 'l\_plist)

- RETURNS l\_plist.
- SIDE EFFECT: The property list of s\_atm is set to l\_plist.

(get 'ls\_name 'g\_ind)

- RETURNS the value under indicator g\_ind in ls\_names property list if ls\_name is a symbol.
- NOTE: If there is no indicator g\_ind in ls\_name's property list, *nil* is returned. If ls\_name is a list of an odd number of elements, then it is a disembodied property list.
- **get** searches a disembodied property list by starting at its **cdr** and comparing every other element with g\_ind, using **eq**.

(getl 'ls\_name 'l\_indicators)

- RETURNS the property list ls\_name beginning at the first indicator that is a member of the list l\_indicators, or *nil*, if none of the indicators in l\_indicators are on ls\_name's property list.
- NOTE: If ls\_name is a list, then it is assumed to be a disembodied property list.

(putprop 'ls\_name 'g\_val 'g\_ind)

(defprop ls\_name g\_val g\_ind)

- RETURNS g\_val.
- SIDE EFFECT: Adds to the property list of ls\_name the value g\_val under the indicator g\_ind.
- NOTE: **putprop** evaluates its arguments; **defprop** does not. ls\_name may be a disembodied property list. See **get**.

(remprop 'ls\_name 'g\_ind)

- RETURNS the portion of ls\_name's property list beginning with the property under the indicator g\_ind. If there is no g\_ind indicator in ls\_name's plist, *nil* is returned.
- SIDE EFFECT: The value under indicator g\_ind and g\_ind itself is removed from the property list of ls\_name.
- NOTE: ls\_name may be a disembodied property list. See **get**.



---

```
=> (putprop 'xlate 'a 'alpha)
a
=> (putprop 'xlate 'b 'beta)
b
=> (plist 'xlate)
(alpha a beta b)
=> (get 'xlate 'alpha)
a
; You can use a disembodied property list this way:
=> (get '(nil fateman rjf sklower kls foderaro jkf) 'sklower)
kls
```

---

### 2.9.3 Tconc structure

A tconc structure is a special type of list, designed to make it easy to add objects to the end. It consists of a list cell whose **car** points to a list of the elements added with **tconc** or **lconc** and whose **cdr** points to the last list cell of the list pointed to by the **car**.

(tconc 'l\_ptr 'g\_x)

- WHERE l\_ptr is a tconc structure.
- RETURNS l\_ptr with g\_x added to the end.

(lconc 'l\_ptr 'l\_x)

- WHERE l\_ptr is a tconc structure.
- RETURNS l\_ptr with the list l\_x spliced in at the end.

---

```
; A tconc structure can be initialized in two ways.
; Nil can be given to tconc, in which case tconc generates
; a tconc structure.
=> (setq foo (tconc nil 1))
((1) 1)

; Since tconc destructively adds to
; the list, you can now add to foo without using setq again.
=> (tconc foo 2)
((1 2) 2)
=> foo
((1 2) 2)

; Another way to create a null tconc structure
; is to use (ncons nil).
=> (setq foo (ncons nil))
(nil)
```

```
=> (tconc foo 1)
((1) 1)

; Now see what lconc can do:
=> (lconc foo nil)
((1) 1) ; There is no change.
=> (lconc foo '(2 3 4))
((1 2 3 4) 4)
```

---

## 2.10 Random functions

The following functions do not fall into any of the classifications above.

**(bcdad 's\_funcname)**

- RETURNS a fixnum that is the address in memory where the function *s\_funcname* begins. If *s\_funcname* is not a machine coded function (binary), then **bcdad** returns *nil*.

**(copy 'g\_arg)**

- RETURNS a structure **equal** to *g\_arg* but with new list cells.

**(copyint\* 'x\_arg)**

- RETURNS a fixnum with the same value as *x\_arg* but in a freshly allocated cell.

**(cpy1 'xvt\_arg)**

- RETURNS a new cell of the same type as *xvt\_arg* with the same value as *xvt\_arg*.

**(getaddress 's\_entry1 's\_binder1 'st\_discipline1 [... ..])**

- RETURNS the binary object that *s\_binder1*'s function field is set to.
- NOTE: This looks in the running lisp's symbol table for a symbol with the same name as *s\_entry1*. It then creates a binary object whose entry field points to *s\_entry1* and whose discipline is *st\_discipline1*. This binary object is stored in the function field of *s\_binder1*. If *st\_discipline1* is *nil*, then "subroutine" is used by default. This is especially useful for **cfasl** users.

**(macroexpand 'g\_form)**

- RETURNS *g\_form* after all macros in it are expanded.



(ptr 'g\_arg)

- RETURNS a value cell initialized to point to g\_arg.

(quote g\_arg)

- RETURNS g\_arg.
- NOTE: The reader allows you to abbreviate (quote foo) as 'foo.

(kwote 'g\_arg)

- RETURNS (list (quote quote) g\_arg).

(replace 'g\_arg1 'g\_arg2)

- WHERE g\_arg1 and g\_arg2 must be the same type of LISPval and not symbols or hunks.
- RETURNS g\_arg2.
- SIDE EFFECT: The effect of **replace** depends on the type of the g\_arg1, although you may notice a similarity in the effects. To understand what **replace** does to fixnum and flonum arguments, you must first understand that such numbers are 'boxed' in FRANZ LISP. This means that if the symbol x has a value 32412, then, in memory, the value element of x's symbol structure contains the address of another word of memory (called a box) with 32412 in it.
  - Thus, there are two ways of changing the value of x. The first way is to change the value element of x's symbol structure to point to a word of memory with a different value. The second way is to change the value in the box that x points to. The former method is used almost all of the time; the latter is used very rarely and may cause great confusion. The function **replace** allows you to do the latter, i.e., to actually change the value in the box.
  - You should watch out for these situations. If you do (setq y x), then both x and y point to the same box. If you now (replace x 12345), then y also has the value 12345. And, in fact, there may be many other pointers to that box.
  - Another problem with replacing fixnums is that some boxes are read-only. The fixnums between -1024 and 1023 are stored in a read-only area, and attempts to replace them result in an "illegal memory reference" error. See the description of **copyint\*** for a way around this problem.
  - For the other valid types, the effect of **replace** is easy to understand. The fields of g\_val1's structure are made **eq** to the corresponding fields of g\_val2's structure. For example, if x and y have lists as values then the effect of (replace x y) is the same as (rplaca x (car y)) and (rplacd x (cdr y)).

(scons 'x\_arg 'bs\_rest)

- WHERE bs\_rest is a bignum or nil.
- RETURNS A bignum whose first 'bigit' (digit in the bignum base) is x\_arg and whose higher order bigits are bs\_rest.

(**sort** 'l\_data 'u\_comparefn)

- RETURNS a list of the elements of l\_data ordered by the comparison function u\_comparefn.
- SIDE EFFECT: The list l\_data is modified rather than allocated in new storage.
- NOTE: (comparefn 'g\_x 'g\_y) should return something non-*nil*, if g\_x can precede g\_y in sorted order; *nil*, if g\_y must precede g\_x.  
If u\_comparefn is *nil*, alphabetical order is used.

(**sortcar** 'l\_list 'u\_comparefn)

- RETURNS a list of the elements of l\_list with the cars ordered by the sort function u\_comparefn.
- SIDE EFFECT: The list l\_list is modified rather than copied.
- NOTE: Like **sort**, if u\_comparefn is *nil*, alphabetical order is used.



The first part of the report is a general description of the project and its objectives. It also includes a brief history of the project and a list of the people involved.

The second part of the report is a detailed description of the project's progress. It includes a list of the tasks that have been completed and a list of the tasks that are still pending.

The third part of the report is a summary of the project's results. It includes a list of the achievements that have been made and a list of the challenges that have been faced.

The fourth part of the report is a conclusion. It includes a list of the main findings of the project and a list of the recommendations that have been made.

The fifth part of the report is a list of references. It includes a list of the books, articles, and other sources that have been used in the project.

The sixth part of the report is a list of appendices. It includes a list of the tables, figures, and other materials that are included in the report.

The seventh part of the report is a list of acknowledgments. It includes a list of the people and organizations that have helped with the project.

The eighth part of the report is a list of the project's budget. It includes a list of the costs that have been incurred and a list of the funds that have been received.

### 3 Arithmetic and logic functions

- 3.1 Simple arithmetic functions 3-1
- 3.2 Predicates 3-3
- 3.3 Trigonometric functions 3-4
- 3.4 Bignum and fixnum manipulation 3-5
- 3.5 Bit manipulation 3-6
- 3.6 Other functions 3-8





## 3 Arithmetic and logic functions

This chapter describes FRANZ LISP's functions for manipulation of numeric quantities. Often the same function is known by several names for the sake of compatibility with several LISP dialects. For example, **add** is also **plus** and **sum**. However, you should avoid using functions with names such as **+** and **\*** in this version of FRANZ LISP unless their arguments are *fixnums*. The **+** function in Common LISP has been defined to allow mixed mode arguments (an incompatibility with preëxisting LISPs such as FRANZ LISP) and corresponds to the FRANZ LISP function **plus**. The mapping of **+** to **plus** and other similar cases may be easily achieved via the package system for Common LISP compatibility. Note that the FRANZ LISP compiler can take advantage of implicit declarations.

An attempt to divide or to generate a floating point result outside of the range of floating point numbers causes a floating exception signal from the operating system. You can catch and process this interrupt if desired. See the description of the **signal** function.

### 3.1 Simple arithmetic functions

```
(add ['n_arg1 ...])  
(plus ['n_arg1 ...])  
(sum ['n_arg1 ...])  
(+ ['x_arg1 ...])
```

- RETURNS the sum of the arguments. If no arguments are given, 0 is returned.
- NOTE: If the size of the partial sum exceeds the limit of a *fixnum*, the partial sum is converted to a *bignum*. If any of the arguments are *flonums*, the partial sum is converted to a *flonum* when that argument is processed and the result is thus a *flonum*. If in the process of doing the addition a *bignum* must be converted into a *flonum*, an error message results.

```
(add1 'n_arg)  
(1+ 'x_arg)
```

- RETURNS its argument plus 1.



(diff ['n\_arg1 ... ])  
(difference ['n\_arg1 ... ])  
(- ['x\_arg1 ... ])

■ RETURNS the result of subtracting from *n\_arg1* all subsequent arguments. If no arguments are given, 0 is returned.

■ NOTE: See the description of **add** for details on data type conversions and restrictions.

(sub1 'n\_arg)  
(1- 'x\_arg)

■ RETURNS its argument minus 1.

(minus 'n\_arg)

■ RETURNS zero minus *n\_arg*.

(product ['n\_arg1 ... ])  
(times ['n\_arg1 ... ])  
(\* ['x\_arg1 ... ])

■ RETURNS the product of all of its arguments. It returns 1 if there are no arguments.

■ NOTE: See the description of the function **add** for details and restrictions to the automatic data type coercion.

(quotient ['n\_arg1 ...])  
(/ ['x\_arg1 ...])

■ RETURNS the result of dividing the first argument by succeeding ones.

■ NOTE: If there are no arguments, 1 is returned. See the description of the function **add** for the details and restrictions of data type coercion. A divide by zero causes a floating exception interrupt. See the description of the **signal** function.

(\*quo 'i\_x 'i\_y)

■ RETURNS the integer part of *i\_x*/*i\_y*.

(Divide 'i\_dividend 'i\_divisor)

■ RETURNS a list whose **car** is the quotient and whose **cadr** is the remainder of the division of *i\_dividend* by *i\_divisor*.

■ NOTE: This is restricted to integer division.

(Emuldiv 'x\_fact1 'x\_fact2 'x\_addn 'x\_divisor)

- RETURNS a list of the quotient and remainder of this operation:  $((x\_fact1 \times x\_fact2) + (\text{sign extended } x\_addn) / x\_divisor$ .

(\*invmod 'x\_number 'x\_modulus)

- RETURNS: This function returns the inverse of *x\_number* in the finite field of the integers modulo the odd prime *x\_modulus*. The result is expressed as a positive or negative *fixnum* of magnitude less than  $x\_modulus / 2$ .
- NOTE: This is useful in algebraic manipulation and number theory calculations.

### 3.2 Predicates

(numberp 'g\_arg)

(numbp 'g\_arg)

- RETURNS *t* iff *g\_arg* is a number: *fixnum*, *flonum*, or *bignum*.

(fixp 'g\_arg)

- RETURNS *t* iff *g\_arg* is a *fixnum* or *bignum*.

(floatp 'g\_arg)

- RETURNS *t* iff *g\_arg* is a *flonum*.

(evenp 'x\_arg)

- RETURNS *t* iff *x\_arg* is even.

(oddp 'x\_arg)

- RETURNS *t* iff *x\_arg* is odd.

(zerop 'g\_arg)

- RETURNS *t* iff *g\_arg* is a number equal to 0.

(onep 'g\_arg)

- RETURNS *t* iff *g\_arg* is a number equal to 1.

(plusp 'n\_arg)

- RETURNS *t* iff *n\_arg* is greater than zero.



(minusp 'g\_arg)

■ RETURNS *t* iff *g\_arg* is a negative number.

(greaterp ['n\_arg1 ...])

(> 'fx\_arg1 'fx\_arg2)

(>& 'x\_arg1 'x\_arg2)

■ RETURNS *t* iff the arguments are in a strictly decreasing order.

■ NOTE: In the functions **greaterp** and **>**, the function **difference** is used to compare adjacent values. If any of the arguments are non-numbers, the error message comes from the **difference** function. The arguments to **>** must be both *fixnums* or both *flonums*. The arguments to **>&** must both be *fixnums*.

(lessp ['n\_arg1 ...])

(< 'fx\_arg1 'fx\_arg2)

(<& 'x\_arg1 'x\_arg2)

■ RETURNS *t* iff the arguments are in a strictly increasing order.

■ NOTE: In functions **lessp** and **<** the function **difference** is used to compare adjacent values. If any of the arguments are not numbers, the error message comes from the **difference** function. The arguments to **<** may be either *fixnums* or *flonums* but must be the same type. The arguments to **<&** must be *fixnums*.

(= 'fx\_arg1 'fx\_arg2)

(=& 'x\_arg1 'x\_arg2)

■ RETURNS *t* iff the arguments have the same value. The arguments to **=** must be the either both *fixnums* or both *flonums*. The arguments to **=&** must be *fixnums*.

(<=& 'x\_arg1 'x\_arg2)

■ EQUIVALENT TO: (not (>& x y)).

(>=& 'x\_arg1 'x\_arg2)

■ EQUIVALENT TO: (not (<& x y)).

(primep 'x\_arg)

■ RETURNS *t* iff *x\_arg* is a prime integer in the range of *fixnums*.

■ NOTE: This is intended to be fast and convenient for, for example, hash-table construction.

### 3.3 Trigonometric functions

Some of these functions are taken from the host math library, and therefore have the same accuracy and other performance characteristics.



(cos 'fx\_angle)

- RETURNS the (*flonum*) cosine of *fx\_angle* (which is assumed to be in radians).

(sin 'fx\_angle)

- RETURNS the sine of *fx\_angle* (which is assumed to be in radians).

(acos 'fx\_arg)

- RETURNS the (*flonum*) arccosine of *fx\_arg* in the range 0 to  $\pi$ .

(asin 'fx\_arg)

- RETURNS the (*flonum*) arcsine of *fx\_arg* in the range  $-\pi/2$  to  $\pi/2$ .

(atan 'fx\_arg1 'fx\_arg2)

- RETURNS the (*flonum*) arctangent of *fx\_arg1* / *fx\_arg2* in the range  $-\pi$  to  $\pi$ .

### 3.4 Bignum and fixnum manipulation

(haipart *bx\_number* *x\_bits*)

- RETURNS a *fixnum* (or *bignum*) that contains the *x\_bits* high bits of (abs *bx\_number*) if *x\_bits* is positive; otherwise, it returns the (abs *x\_bits*) low bits of (abs *bx\_number*).

(haulong 'bx\_number)

(integer-length 'bx\_number)

- RETURNS the number of significant bits in *bx\_number*.
- NOTE: The result is equal to the least integer greater than or equal to the base two logarithm of one plus the absolute value of *bx\_number*. Common LISP's integer-length differs from haulong in that if the argument is negative, the result is equal to the least integer greater than or equal to the base two logarithm of the absolute value of *bx\_number*.

(bignum-leftshift *bx\_arg* *x\_amount*)

- RETURNS *bx\_arg* shifted left by *x\_amount*. If *x\_amount* is negative, *bx\_arg* is shifted right by the magnitude of *x\_amount*.
- NOTE: If *bx\_arg* is shifted right, it will be rounded to the nearest even number.

(sticky-bignum-leftshift 'bx\_arg 'x\_amount)

- RETURNS *bx\_arg* shifted left by *x\_amount*. If *x\_amount* is negative, *bx\_arg* will be shifted right by the magnitude of *x\_amount* and rounded.



■ NOTE: Sticky rounding is done this way: after shifting, the low order bit is changed to 1 if any 1's were shifted off to the right.

### 3.5 Bit manipulation

In this section numerous functions based on the **boole** function are defined for Common LISP compatibility. Many are merely macro-expanded into calls to **boole**, and in the current implementation are restricted to *fixnum*-length integers.

(**boole** 'x\_key 'x\_v1 'x\_v2 ...)

■ RETURNS the result of the bitwise boolean operation as described in the following table.

■ NOTE: If there are more than 3 arguments, then evaluation proceeds left to right with each partial result becoming the new value of x\_v1. That is,

(**boole** 'key 'v1 'v2 'v3)  $\equiv$  (**boole** 'key (**boole** 'key 'v1 'v2) 'v3)

In the following table,  $\times$  represents bitwise and,  $+$  represents bitwise or,  $\neq$  represents bitwise exclusive-or, and  $\neg$  represents bitwise negation. Negation is the highest precedence operator.

(boole 'key 'x 'y)								
key	0	1	2	3	4	5	6	7
result	0	$x \times y$	$\neg x \times y$	$y$	$x \times \neg y$	$x$	$x \neq y$	$x + y$
name		and			bitclear		xor	or
key	8	9	10	11	12	13	14	15
result	$\neg(x + y)$	$\neg(x \neq y)$	$\neg x$	$\neg x + y$	$\neg y$	$x + \neg y$	$\neg x + \neg y$	-1
names	nor	equiv		implies			nand	

(**lsh** 'x\_val 'x\_amt)

(**ash** 'x\_val 'x\_amt)

■ RETURNS x\_val shifted left by x\_amt if x\_amt is positive. If x\_amt is negative, then **lsh** returns x\_val shifted right by the magnitude if x\_amt.

■ NOTE: This always returns a *fixnum* even for those numbers whose magnitude is so large that they would normally be represented as a *bignum*; i.e., shifter bits are lost. Functionally "arithmetic shift" **ash**, is the same as **lsh**. For more general bit shifters, see **bignum-leftshift** and **sticky-bignum-leftshift**.



(rot 'x\_val 'x\_amt)

- RETURNS *x\_val* rotated left by *x\_amt* if *x\_amt* is positive. If *x\_amt* is negative, then *x\_val* is rotated right by the magnitude of *x\_amt*.

(logand ['n\_arg1 ...])

- RETURNS the (numeric) result of the bitwise logical *and* of the *n\_argis*.
- NOTE: If there are no arguments, *-1* is returned.

(logior ['n\_arg1 ...])

- RETURNS the (numeric) result of the bitwise logical *inclusive or* of the *n\_argis*.
- NOTE: If there are no arguments, zero is returned.

(logxor ['n\_arg1 ...])

- RETURNS the (numeric) result of the bitwise logical *exclusive or* of the *n\_argis*.
- NOTE: If there are no arguments, zero is returned.

(logeqv ['n\_arg1 ...])

- RETURNS the (numeric) result of the bitwise logical *equivalence* of the *n\_argis*.
- NOTE: If there are no arguments, *-1* is returned. Equivalence is the same as *exclusive nor*.

(lognot 'n\_arg)

- RETURNS the (numeric) result of the bitwise logical *not* of *n\_arg*. (The logical complement of *n\_arg*.)

Additional Common LISP logical operators which require exactly two *fixnum* arguments follow.

(logandc1 'n\_arg1 'n\_arg2)

- RETURNS (logand (lognot *n\_arg1*) *n\_arg2*).

(logandc2 'n\_arg1 'n\_arg2)

- RETURNS (logand *n\_arg1* (lognot *n\_arg2*)).

(logbitp 'x\_index 'n\_number)

- RETURNS *t* if the bit in *n\_number* with index *x\_index* is one. (logbitp 0 *n\_number*) is true if *n\_number* is a positive odd number.



(logcount 'n\_number)

- RETURNS the number of one bits in n\_number if n\_number is positive; the number of zero bits if n\_number is negative.

(logiorc1 'n\_arg1 'n\_arg2)

- RETURNS (logior (lognot n\_arg1) n\_arg2).

(logiorc2 'n\_arg1 'n\_arg2)

- RETURNS (logior n\_arg1 (lognot n\_arg2)).

(lognand 'n\_arg1 'n\_arg2)

- RETURNS (lognot (logand n\_arg1 n\_arg2)).

(lognor 'n\_arg1 'n\_arg2)

- RETURNS (lognot (logor n\_arg1 n\_arg2)).

(logtest 'n\_arg1 'n\_arg2)

- RETURNS (not (zerop (logand n\_arg1 n\_arg2))).
- NOTE: Logtest is true if any one's are in corresponding locations in the two arguments.

### 3.6 Other functions

As noted above, some of the following functions are inherited from the host math library.

(abs 'n\_arg)

(absval 'n\_arg)

- RETURNS the absolute value of n\_arg.

(exp 'fx\_arg)

- RETURNS  $e$  raised to the fx\_arg power (*flonum*).

(expt 'n\_base 'n\_power)

- RETURNS n\_base raised to the n\_power power.
- NOTE: If either of the arguments are *flonums*, the calculation is done using *log* and *exp*.

(fact 'x\_arg)

- RETURNS *x\_arg* factorial—*fixnum* or *bignum*.

(fix 'n\_arg)

- RETURNS a *fixnum* as close as we can get to *n\_arg*.
- NOTE: *fix* rounds down. Currently, if *n\_arg* is a *flonum* larger than the size of a *fixnum*, this fails.

(float 'n\_arg)

- RETURNS a *flonum* as close as we can get to *n\_arg*.
- NOTE: If *n\_arg* is a *bignum* larger than the maximum size of a *flonum*, then a floating exception occurs.

(log 'fx\_arg)

- RETURNS the natural logarithm of *fx\_arg*.

(max 'n\_arg1 ... )

- RETURNS the maximum value in the list of arguments.

(min 'n\_arg1 ... )

- RETURNS the minimum value in the list of arguments.

(mod 'i\_dividend 'i\_divisor)

(remainder 'i\_dividend 'i\_divisor)

- RETURNS the remainder when *i\_dividend* is divided by *i\_divisor*.
- NOTE: The sign of the result has the same sign as *i\_dividend*.

(\*mod 'x\_dividend 'x\_divisor)

- RETURNS the balanced representation of *x\_dividend* modulo *x\_divisor*.
- NOTE: The range of the balanced representation is  $\text{abs}(x\_divisor)/2$  to  $(\text{abs}(x\_divisor)/2) - x\_divisor + 1$ .

(random ['x\_limit])

- RETURNS a *fixnum* between 0 and *x\_limit* -1 if *x\_limit* is given. If *x\_limit* is not given, any *fixnum*, positive or negative, might be returned.

(sqrt 'fx\_arg)

- RETURNS the square root of *fx\_arg*.



10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

10-10-10

## 4 Special functions

- 4.1 Introduction 4-1
- 4.2 Functions 4-1
- 4.3 Multiple value returns 4-20





## 4 Special functions

### 4.1 Introduction

This chapter describes the special functions, or forms of FRANZ LISP. While LISP is generally thought of as very simple, in fact serious programming in LISP uses a large helping of these special forms. What makes them special is that they generally do not conform to the usual function evaluation methodology.

### 4.2 Functions

(and [g\_arg1 ...])

- RETURNS the value of the last argument if all arguments evaluate to a non-*nil* value; otherwise, and returns *nil*. It returns *t* if there are no arguments.
- NOTE: The arguments are evaluated left to right and evaluation ceases with the first *nil* encountered.

(apply 'u\_func [g\_arg1 ...] 'l\_args)

- RETURNS the result of applying the function *u\_func* to the arguments obtained by appending the list *l\_args* to any optional arguments *g\_argn*.
- NOTE: The arguments to the function *u\_func* are obtained from the arguments to **apply** by splicing in the final argument list. If *u\_func* is a *lambda*, then the length of *l\_args* plus the number of optional arguments must correspond to the number of formal parameters for the function. If *u\_func* is an *nlambda* or *macro*, then *l\_args* is bound to the single formal parameter and no optional arguments are permitted. Further, a *macro* is expanded by **apply** with the argument list, but the expanded form is not evaluated. The function **funcall** evaluates *macros* (see below). If *foo* is a *lambda*, *lexpr*, or *array*, then (apply 'foo 'a '(b c)) is equivalent to (foo 'a 'b 'c). If *foo* is an *nlambda*, then (apply 'foo '(a b c)) is equivalent to (foo a b c). Finally, if *foo* is a *macro*, then (apply 'foo '(foo a b c)) is equivalent to (macroexpand '(foo a b c)).
- This version of **apply** (Opus 42 and later) is upward compatible with Common LISP: it can apply *nlambdas* and *macros*, and the treatment of *lambda* functions is identical.



---

```
; add1 is a lambda of 1 argument
=> (apply 'add1 '(3))
4

; plus takes a number of arguments
=> (apply 'plus 1 2 '(3 4 5))
15
; The above is identical to
=> (plus 1 2 3 4 5)
15

; You can define plus1 as a macro that is equivalent to
; add1.
=> (def plus1 (macro (arg) (list 'add1 (cadr arg))))
plus1
=> (plus1 3)
4

; Now if you apply a macro, you obtain the form it
; changes to.
=> (apply 'plus1 '(plus1 3))
(add1 3)

; If you funcall a macro ,however, the result of the macro
; is eval'd before it is returned.
=> (funcall 'plus1 '(plus1 3))
4

; For this particular macro, the car of the arg is not checked
; so that this too works.
=> (apply 'plus1 '(foo 3))
(add1 3)
```

---

(apropos 'st\_arg ['g\_package])

■ RETURNS *nil*

■ NOTE: All packages are searched for symbols whose print name contains the substring *st\_arg*. Matching symbols are printed, along with information about their function definition and current value. If *g\_package* is specified and is a package, then only that package is searched. If *g\_package* is a symbol or a string then only (find-package *g\_package*) is searched.

(apropos-list 'st\_arg ['g\_package])

■ RETURNS a list of symbols which is the result of searching all packages for symbols which have a print name containing the substring *st\_arg*. If *g\_package* is specified and is a package, then only that package is searched. If *g\_package* is a symbol



or a string then only (find-package g\_package) is searched.

(arg ['x\_numb])

■ RETURNS if x\_numb is specified, then the x\_numbth argument to the enclosing *lexpr*. If x\_numb is not specified, then this returns the number of arguments to the enclosing *lexpr*.

■ NOTE: It is an error to the interpreter if x\_numb is given and out of range.

(break [g\_message [g\_pred]])

■ WHERE if g\_message is not given, it is assumed to be the null string, and if g\_pred is not given, it is assumed to be *t*.

■ RETURNS the value of (\*break 'g\_pred 'g\_message).

(\*break 'g\_pred 'g\_message)

■ RETURNS *nil* immediately if g\_pred is *nil*; otherwise, the value of the next (return 'value) expression typed in at top level.

■ SIDE EFFECT: If the predicate, g\_pred, evaluates to non-*nil*, the LISP system stops and prints out "Break " followed by g\_message. It then enters a break loop that allows you to interactively debug a program. To continue execution from a break, you can use the **return** function. You can also use the **?ret** top level command. See chapter 13 for details on the top level.

(caseq 'g\_key-form l\_clause1 ...)

(case 'g\_key-form l\_clause1 ...)

■ WHERE l\_clause*i* is a list of the form (g\_comparator ['g\_formi ...]). The comparators may be symbols, small *fixnums*, a list of small *fixnums* or symbols.

■ NOTE: The way **caseq** works is that it evaluates g\_key-form, yielding a value called the selector. Each clause is examined until the selector is found consistent with the comparator. For a symbol, or a *fixnum*, this means the two must be **eq**. For a list, this means that the selector must be **eq** to some element of the list.

□ The comparator consisting of the symbol *t* has special semantics: it matches anything and, consequently, should be the last comparator.

□ In any case, having chosen a clause, **caseq** evaluates each form within that clause and returns the value of the last form.

■ RETURNS the value of the last form as indicated above. If no comparators are matched, **caseq** returns *nil*.

---

Here are two ways of defining the same function:

```
=> (defun fate (personna)
      (caseq personna
        (cow '(jumped over the moon))
```



```

      (cat '(played nero))
      ((dish spoon) '(ran away with each other))
      (t '(lived happily ever after)))
fate
=> (defun fate (personna)
      (cond
        ((eq personna 'cow) '(jumped over the moon))
        ((eq personna 'cat) '(played nero))
        ((memq personna '(dish spoon))
          '(ran away with each other))
        (t '(lived happily ever after))))
fate

```

---

(catch g\_exp [ls\_tag])

- WHERE if *ls\_tag* is not given, it is assumed to be *nil*.
- RETURNS the result of (\*catch 'ls\_tag g\_exp).
- NOTE: **catch** is defined as a *macro*.

(\*catch 'ls\_tag g\_exp)

- WHERE *ls\_tag* is either a symbol or a list of symbols.
- RETURNS the result of evaluating *g\_exp* or, if the **throw** pseudo-function is invoked with the argument *ls\_tag* within the execution of *g\_exp*, the value given by **throw**. (See **throw** and \***throw**.) The **\*catch** and **throw** or \***throw** construction is used for a non-local return of a value, and is typically used in an error return or some kind of break in the normal modularization of a program.
- SIDE EFFECT: This proceeds as follows: **\*catch** first sets up a 'catch frame' on the LISP runtime stack. Then it begins to evaluate *g\_exp*. If *g\_exp* evaluates normally, its value is returned. If, however, a value is thrown during the evaluation of *g\_exp*, then this **\*catch** returns with that value if one of these cases is true:
  - The tag thrown to is *ls\_tag*.
  - *ls\_tag* is a list and the tag thrown to is a member of this list.
  - *ls\_tag* is *nil*.
- NOTE: Errors are implemented as a special kind of **throw**. A **catch** with no tag does not catch an error, but a **catch** whose tag is the error type catches that type of error. See Chapter 10 for more information.

(comment [g\_arg ...])

- RETURNS the symbol comment.
- NOTE: This does nothing but return a constant value. You should be caution to avoid using this in a place where the value might be used.



(**cond** [*l\_clause1* ...])  
 (**when** *pred* *form1* ...)  
 (**unless** *pred* *form1* ...)  
 (**if** *pred* *form1* [*form2*])

■ RETURNS the last value evaluated in the first satisfied clause. If no clauses are satisfied, then *nil* is returned.

■ NOTE: **cond** is the basic conditional 'statement' in LISP. The clauses are processed from left to right. The first element of a clause is evaluated. If it evaluates to a non-*nil* value, then that clause is satisfied and all following elements of that clause are evaluated. The last value computed is returned as the value of the **cond**. If there is just one element in the clause, then its value is returned. If the first element of a clause evaluates to *nil*, then the other elements of that clause are not evaluated and the system moves to the next clause. The forms **when**, **unless**, and **if** are macros which expand into **conds** as follows:

(**when** *p a b* ... )  $\equiv$  (**cond** (*p a b* ...))  
 (**unless** *p a b* ... )  $\equiv$  (**cond** ((**not** *p*) *a b* ...))  
 (**if** *p a b*)  $\equiv$  (**cond** (*p a*) (*t b*))

#### (**cvttointlisp**)

■ SIDE EFFECT: The reader is modified to conform with the InterLISP syntax. The character % is made the escape character and special meanings for comma, backquote, and backslash are removed. Also the reader is told to convert upper case to lower case.

#### (**cvttofranzlisp**)

■ SIDE EFFECT: FRANZ LISP's default syntax is reinstated. You should run this function after having run any of the other **cvtto** functions. Backslash is made the escape character, super-brackets work again, and the reader distinguishes between upper and lower case.

#### (**cvttomaclisp**)

■ SIDE EFFECT: The reader is modified to conform with MacLISP syntax. The character / is made the escape character, and the special meanings for backslash, left and right bracket are removed. The reader is made case-insensitive.

#### (**cvttoucillisp**)

■ SIDE EFFECT: The reader is modified to conform with UCI LISP syntax. The character / is made the escape character; tilde is made the comment character; exclamation point takes on the unquote function normally held by comma, and backslash, comma, and semicolon become normal characters. Here too, the reader is made case-insensitive.



(debug s\_msg)

- SIDE EFFECT: Enter the Fixit package described in Chapter 15. This package allows you to examine the evaluation stack in detail. To leave the Fixit package type *ok*.

(debugging 'g\_arg)

- SIDE EFFECT: If *g\_arg* is non-*nil*, FRANZ LISP unlinks the transfer tables, does a (\*rset t) to turn on evaluation monitoring and sets the all-error catcher (*ER%all*) to be *tpl-err-all-fcn*. If *g\_arg* is *nil*, all of the earlier changes are undone.

(declare [g\_arg ...])

- RETURNS *nil*
- NOTE: This is a no-op to the evaluator. It has special meaning to the compiler (see Chapter 12).

(def s\_name (s\_type l\_argl g\_exp1 ...))

- WHERE *s\_type* is one of *lambda*, *nlambda*, *macro* or *lexpr*.
- RETURNS *s\_name*
- SIDE EFFECT: This defines the function *s\_name* to the LISP system. If *s\_type* is *nlambda* or *macro* then the argument list *l\_argl* must contain exactly one non-*nil* symbol.
- If *s\_type* is *macro*, the single argument to the defined *macro* will be bound to the list containing the *macro*'s name and its arguments.

(defmacro s\_name l\_arg g\_exp1 ...)

(defcmacro s\_name l\_arg g\_exp1 ...)

- RETURNS *s\_name*
- SIDE EFFECT: This defines the *macro* *s\_name*. **defmacro** makes it easy to write *macros* since it makes the syntax, including the use of optional parameters *&optional*, *&rest*, *&aux*, and *&key*, just like **defun**. Further information on **defmacro** is in §8.3.2. **defcmacro** defines compiler-only *macros*, or *cmacros*. A *cmacro* is stored on the property list of a symbol under the indicator *cmacro*. Thus a function can have a normal definition and a *cmacro* definition.

(defsubst s\_name l\_list g\_form [...])

- RETURNS *s\_name*.
- NOTE: **defsubst** is like **defun** (below), except that a compiler *macro* is defined (see **defcmacro** above) which expands to the definition of *s\_name* as a *lambda*. Its use is to allow an easy method of defining functions which are to be *macro* expanded out at compile time. One can also use the optional parameters *&optional*, *&rest*, *&aux*, and *&key* just like **defun**.



```
(defun s_name [s_mtype] ls_arg1 g_exp1 ... )
```

- WHERE *s\_mtype* is one of *fexpr*, *expr*, *args* or *macro*.
- RETURNS *s\_name*
- SIDE EFFECT: This defines the function *s\_name*.
- NOTE: Some of these options exist for MacLISP compatibility. **defun** rearranges the information you provide and then invokes the **def** function. For example, the MacLISP *fexpr* is automatically converted to a FRANZ LISP *nlambda*. A MacLISP *s\_mtype* of *expr* is simply the same thing as FRANZ LISP's *lambda*. The *s\_type* of *macro* exists in the same form. If *ls\_arg1* is a non-*nil* symbol, then the type is assumed to be *lexpr* and *ls\_arg1* is the symbol that is bound to the number of args when the function is entered.
- For compatibility with the ZetaLISP, there are four types of optional parameters that can occur in *ls\_arg1*: *&optional*, *&rest*, *&aux* and *&key*. For an explanation of use of these forms, see §8.2.
- An additional form accepted by **defun** is provided to set up property lists. To place the value *g\_value* on the property list of symbol *s\_name* under indicator *s\_ind*, use the form

```
(defun (s_name s_ind) g_value).
```

---

```
; def and defun here are used to define identical functions.
; You can decide for yourself which is easier to use.
```

```
=> (def append1
      (lambda (lis extra)
        (append lis (list extra))))
append1
```

```
=> (defun append1 (lis extra)
      (append lis (list extra)))
append1
```

```
; Using the & forms . . .
```

```
=> (defun test (a b
               &optional c
               &aux (retval 0))
      (if c then
          (msg "Optional arg present" N
               "c is " c N))
          (msg "retval is " retval N))
```

```
test
=> (test 1 2 3 4)
Optional arg present
c is 3
retval is 0
```

---



(defvar s\_variable [g\_init])

- RETURNS *s\_variable*.
- NOTE: This form is put at the top level in files, like **defun**.
- SIDE EFFECT: This declares *s\_variable* to be special. If *g\_init* is present and *s\_variable* is unbound when the file is read in, *s\_variable* is set to the value of *g\_init*. An advantage of (defvar foo) over (declare (special foo)) is that if a file containing **defvars** is loaded (or **fasled**) in during compilation, the variables mentioned in the **defvars** are declared special. The only way to have that effect with (declare (special foo)) is to **include** the file.

(do l\_vrbs l\_test g\_exp1 ...)

- RETURNS the last form in the **cdr** of *l\_test* evaluated, or a value explicitly given by a return evaluated within the **do** body.
- NOTE: This is the basic iteration form for FRANZ LISP. *l\_vrbs* is a list of zero or more var-init-repeat forms. A var-init-repeat form looks like:

(s\_name [g\_init [g\_repeat]]).

- There are three cases depending on what is present in the form. If just *s\_name* is present, this means that when the **do** is entered, *s\_name* is *lambda*-bound to *nil* and is never modified by the system (though the program is certainly free to modify its value). If the form is (s\_name 'g\_init) then the only difference is that *s\_name* is *lambda*-bound to the value of *g\_init* instead of *nil*. If *g\_repeat* is also present then *s\_name* is *lambda*-bound to *g\_init* when the loop is entered and after each pass through the **do** body *s\_name* is bound to the value of *g\_repeat*.
- *l\_test* is either *nil* or has the form of a **cond** clause. If it is *nil* then the **do** body is evaluated only once and the **do** returns *nil*. Otherwise, before the **do** body is evaluated the **car** of *l\_test* is evaluated, and, if the result is non-*nil*, this signals an end to the looping. Then the rest of the forms in *l\_test* are evaluated and the value of the last one is returned as the value of the **do**. If the **cdr** of *l\_test* is *nil*, then *nil* is returned. Thus, this is not exactly like a **cond** clause.
- *g\_exp1* and those forms that follow constitute the **do** body. A **do** body is like a **prog** body and, thus, may have labels. You can use the functions **go** and **return**.
- The sequence of evaluations is this:
  - 1 The init forms are evaluated left to right and stored in temporary locations.
  - 2 Simultaneously, all **do** variables are *lambda* bound to the value of their init forms or to *nil*.
  - 3 If *l\_test* is non-*nil*, then the **car** is evaluated, and, if it is non-*nil*, the rest of the forms in *l\_test* are evaluated, and the last value is returned as the value of the **do**.
  - 4 The forms in the **do** body are evaluated left to right.
  - 5 If *l\_test* is *nil* the **do** function returns with the value *nil*.
  - 6 The repeat forms are evaluated and saved in temporary locations.
  - 7 The variables with repeat forms are simultaneously bound to the values of those forms.



□ 8 Go to step 3.

■ NOTE: There is an alternate form of **do** that can be used when there is only one **do** variable. It is described next.

□ For Common LISP compatibility, the var-init-repeat form may be an atom, which is equivalent to specifying a **g\_init** of *nil*.

(**do** s\_name g\_init g\_repeat g\_test g\_exp1 ...)

■ NOTE: This is another, less general, form of **do**. It is evaluated by:

□ 1 Evaluating **g\_init**.

□ 2 *Lambda* binding s\_name to value of **g\_init**.

□ 3 **g\_test** is evaluated, and, if it is not *nil*, the **do** function returns with *nil*.

□ 4 The **do** body is evaluated beginning at **g\_exp1**.

□ 5 The repeat form is evaluated and stored in s\_name.

□ 6 Go to step 3.

■ RETURNS *nil*.

(**do\*** l\_vrbs l\_test g\_exp1 ... )

■ RETURNS the value of the last form in the **cdr** of **l\_test**, or a value explicitly given by a **return** evaluated within the **do** body.

■ NOTE: This is the Common LISP **do\*** form. It is very similar to **do** except that: (1) The var-init-repeat forms are evaluated sequentially rather than simultaneously. (2) There is no analogue of the old-style MacLISP **do**. In particular, **l\_vrbs** must be a list of var-init-repeat forms.

(**dolist** (s\_var l\_form g\_resultform) g\_exp1 ...)

■ RETURNS if present, **g\_resultform**, *nil* otherwise. **dolist** provides a mechanism to iterate over the elements of the list **l\_form**, successively binding the elements to **s\_var**, while executing the body of the loop **g\_form**.

(**dotimes** (s\_var i\_countform g\_resultform) g\_exp1 ...)

■ RETURNS if present, **g\_resultform**, *nil* otherwise. **dotimes** provides a mechanism to iterate over a sequence of integers. First, **i\_countform** is evaluated to produce an integer, and then evaluates **g\_form** once for each integer from zero (inclusive) to **i\_countform** (exclusive), in order, binding **s\_var** to this integer.

---

```
; This is a simple function that numbers the elements of a
; list. It uses a do function with two local variables.
=> (defun printem (lis)
      (do ((xx lis (cdr xx))
          (i 1 (1+ i)))
          ((null xx) (atom "all done") (terpr))
        (print i)))
```



```

      (patom ": ")
      (print (car xx))
      (terpr))
printem
=> (printem '(a b c d))
1: a
2: b
3: c
4: d
all done
nil
=> (setq x 100)
100
=> (dolist (x '(a b c d e f g) 'result)
      (msg x " "))
a b c d e f g result
=> x
100
=> (dotimes (x 10) (msg x ","))
0,1,2,3,4,5,6,7,8,9,nil
=> x
100

```

---

(environment [*l\_when1 l\_what1 l\_when2 l\_what2 ...*])  
 (environment-maclisp [*l\_when1 l\_what1 l\_when2 l\_what2 ...*])  
 (environment-lmlisp [*l\_when1 l\_what1 l\_when2 l\_what2 ...*])

■ WHERE the *l\_wheni* are a subset of (eval compile load), and the symbols have the same meaning as they do in 'eval-when.'

□ The *l\_whati* may be:

(files file1 file2 ... file*n*)

which insure that the named files are loaded. To see if file*i* is loaded, these functions look for a 'version' property under file*i*'s property list. In order to make this work to prevent multiple loading, you should put

(putprop 'myfile t 'version)

at the end of 'myfile.l.'

□ Another acceptable form for an *l\_whati* is

(syntax type)

where type is either *maclisp*, *intlisp*, *ucilisp*, or *franzlisp*.

■ SIDE EFFECT: **environment-maclisp** sets the environment to what the command 'liszt -m' generates.

□ **environment-lmlisp** sets up the LISP Machine environment. This is like *maclisp* but it has additional *macros*.



- For these specialized environments, only the *files* clauses are useful.
- **EXAMPLE:** (environment-maclisp (compile eval) (files foo bar)).
- **RETURNS** the last list of files requested.

(err ['s\_value [nil]])

- **RETURNS** nothing (it never returns).
- **SIDE EFFECT:** This causes an error, and, if this error is caught by an **errset** then that **errset** returns *s\_value* instead of *nil*. If the second arg is given, then it must be *nil* (for MacLISP compatibility).

(error ['s\_message1 ['s\_message2]])

- **RETURNS** nothing (it never returns).
- **SIDE EFFECT:** *s\_message1* and *s\_message2* are **patomed** if they are given and then **err** is called (with no arguments), which causes an error.

(cli:error 's\_format-string ['args])

- **WHERE** *s\_format-string* is a command string for **format**, and the *args* are passed to **format** as parameters for the command string.
- **RETURNS** nothing. This function signals a fatal error.
- **NOTE:** The prefix *cli:* must be used normally because it is part of the Common LISP package, and conflicts with the FRANZ LISP function of the same name.

(cerror 's\_continue-format-string 's\_error-format-string ['arg ...])

- **RETURNS** *nil*, if the program is continued after the error.
- **SIDE EFFECT:** This function (which is intended for use with continuable errors) signals an error and enters a break loop. The program may be continued after resolution of the error by typing *?ret* to the break loop. The two strings given are intended as control strings to the **format** function to construct a continuation message and an error message. This is entirely compatible with the Common LISP **cerror** function.

(errset g\_expr [s\_flag])

- **RETURNS** a list of one element that is the value resulting from evaluating *g\_expr*. If an error occurs during the evaluation of *g\_expr*, then the locus of control returns to the **errset**, which then returns *nil* (unless the error was caused by a call to **err** with a non-*nil* argument).
- **SIDE EFFECT:** *s\_flag* is evaluated before *g\_expr* is evaluated. If *s\_flag* is not given, then it is assumed to be *t*. If an error occurs during the evaluation of *g\_expr*, and *s\_flag* was evaluated to a non-*nil* value, then the error message associated with the error is printed before control returns to **errset**.



(eval 'g\_val ['x\_bind-pointer])

- RETURNS the result of evaluating g\_val.
- NOTE: The evaluator evaluates g\_val in the following way:
  - If g\_val is a symbol, then the evaluator returns its value. If g\_val had never been assigned a value, then this causes an 'unbound variable' error. If x\_bind-pointer is given, then the variable is evaluated with respect to that pointer. See **evalframe** for details on bind-pointers.
  - If g\_val is of type value, then its value is returned. If g\_val is of any other type than list, g\_val is returned.
  - If g\_val is a list object, then g\_val is either a function call or array reference. Let g\_car be the first element of g\_val. g\_car is continually evaluated until it results in a symbol with a non-*nil* function binding or a non-symbol. Call the result: g\_func.
  - g\_func must be one of three types: list, binary, or array. If it is a list, then the first element of the list, which is called g\_func<sub>type</sub>, must be either *lambda*, *nlambda*, *macro*, or *lexpr*. If g\_func is a binary, then its discipline, which is called g\_func<sub>type</sub>, is either *lambda*, *nlambda*, *macro*, or a string. If g\_func is an array, then this form is evaluated specially. See Chapter 9 on arrays. If g\_func is a list or binary, then g\_func<sub>type</sub> determines how the arguments to this function, the **cdr** of g\_val, are processed. If g\_func<sub>type</sub> is a string, then this is a foreign function call. See §18.3 for more details.
  - If g\_func<sub>type</sub> is *lambda* or *lexpr*, the arguments are evaluated (by calling **eval** recursively) and stacked. If g\_func<sub>type</sub> is *nlambda*, then the argument list is stacked. If g\_func<sub>type</sub> is *macro*, then the entire form, g\_val, is stacked.
  - Next, the formal variables are *lambda* bound. The formal variables are the **cadr** of g\_func. If g\_func<sub>type</sub> is *nlambda*, *lexpr*, or *macro*, there should only be one formal variable. The values on the stack are *lambda* bound to the formal variables except in the case of a *lexpr*, where the number of actual arguments is bound to the formal variable.
  - After the binding is done, the function is invoked, either by jumping to the entry point in the case of a binary or by evaluating the list of forms beginning at (**cddr** g\_func). The result of this function invocation is returned as the value of the call to **eval**.

(evalframe 'x\_pdlpointer)

- RETURNS an **evalframe** descriptor for the evaluation frame just before x\_pdlpointer. If x\_pdlpointer is *nil*, it returns the evaluation frame of the frame just before the current call to **evalframe**.
- NOTE: An **evalframe** descriptor describes a call to **eval**, **apply**, or **funcall**. The form of the descriptor is

(type pdl-pointer expression bind-pointer np-index lbot-index)

where type is *eval* if this describes a call to **eval** or **apply** if this is a call to **apply** or **funcall**. pdl-pointer is a number that describes this context. It can be passed to



**evalframe** to obtain the next descriptor and can be passed to **freturn** to cause a return from this context. **bind-pointer** is the size of variable binding stack when this evaluation began. The **bind-pointer** can be given as a second argument to **eval** in order to evaluate variables in the same context as this evaluation. If type is *eval*, then expression has the form (function-name arg1 ...). If type is *apply*, then expression has the form (function-name (arg1 ...)). **np-index** and **lbot-index** are pointers into the argument stack (also known as the **namestack** array) at the time of call. **lbot-index** points to the first argument; **np-index** points one beyond the last argument.

□ In order for there to be enough information for **evalframe** to return, you must call (**\*rset** t).

■ **EXAMPLE:** (progn (evalframe nil)) returns (eval 2147478600 (progn (evalframe nil)) 1 8 7).

(**evalhook** 'g\_form 'su\_evalfunc ['su\_funcallfunc])

■ **RETURNS** the result of evaluating **g\_form** after *lambda*-binding **evalhook** to **su\_evalfunc**, and, if it is given, *lambda*-binding **funcallhook** to **su\_funcallhook**.

■ **NOTE:** As explained in §15.4, the function **eval** may pass the job of evaluating a form to a user *hook* function when various switches are set. The hook function normally prints the form to be evaluated on the terminal and then evaluates it by calling **evalhook**. **evalhook** does the *lambda* binding mentioned earlier and then calls **eval** to evaluate the form after setting an internal switch to tell **eval** not to call the user's hook function just this one time. This allows the evaluation process to advance one step and yet insure that further calls to **eval** cause traps to the hook function (if **su\_evalfunc** is non-*nil*).

□ In order for **evalhook** to work, (**\*rset** t) and (**sstatus evalhook** t) must have been done previously.

(**freturn** 'x\_pdl-pointer 'g\_retval)

■ **RETURNS** **g\_retval** from the context given by **x\_pdl-pointer**.

■ **NOTE:** A *pdl-pointer* denotes a certain expression currently being evaluated. The *pdl-pointer* for a given expression can be obtained from **evalframe**.

(**funcall** 'u\_func [g\_arg1 ...])

■ **RETURNS** the value of applying function **u\_func** to the arguments **g\_argi** and then evaluating that result if **u\_func** is a *macro*.

■ **NOTE:** If **u\_func** is a *macro* or *nlambda*, then there should be only one **g\_arg**, which should be a list. **funcall** is the function that the evaluator uses to evaluate lists. If **foo** is a *lambda*, *lexpr*, or array, then (funcall 'foo 'a 'b 'c) is equivalent to (foo 'a 'b 'c). If **foo** is an *nlambda*, then (funcall 'foo '(a b c)) is equivalent to (foo a b c). Finally, if **foo** is a *macro*, then (funcall 'foo '(foo a b c)) is equivalent to (foo a b c).



(funcallhook 'l\_form 'su\_funcallfunc ['su\_evalfunc])

■ RETURNS the result of **funcall** applying (car l\_form) to the evaluated arguments of (cdr l\_form), after funcallhook is *lambda*-bound to su\_funcallfunc, and evalhook is *lambda*-bound to su\_evalfunc if the argument is present.

■ NOTE: This function is designed to continue the evaluation process with as little work as possible after a **funcallhook** trap has occurred. It is for this reason that the form of l\_form is unorthodox: its **car** is the name of the function to call and its **cdr** are a list of arguments to stack (without evaluating again) before calling the given function. After stacking the arguments but before calling **funcall**, an internal switch is set to prevent **funcall** from passing the job of **funcalling** to su\_funcallfunc. If **funcall** is called recursively in **funcalling** l\_form and if su\_funcallfunc is non-*nil*, then the arguments to **funcall** are actually given to su\_funcallfunc (a *lexpr*) to be **funcalled**.

□ In order for **funcallhook** to work, (\*rset t) and (sstatus funcallhook t) must have been done previously. A more detailed description of **evalhook** and **funcallhook** is given in Chapter 15.

(function u\_func)

■ RETURNS the function binding of u\_func if it is a symbol with a function binding; otherwise, u\_func is returned.

(go g\_labexp)

■ WHERE g\_labexp is either a symbol or an expression.

■ SIDE EFFECT: If g\_labexp is an expression, that expression is evaluated and should result in a symbol. The locus of control moves to just following the symbol g\_labexp in the current **prog** or **do** body.

■ NOTE: This is only valid in the context of a **prog** or **do** body. The interpreter and compiler allow non-local **gos**, although the compiler does not allow a **go** to leave a function body. The compiler does not allow g\_labexp to be an expression.

(help sx\_arg)

■ RETURNS *nil*.

■ NOTE: a portion of the on-line FRANZ LISP manual is printed. If sx\_arg is a function, the manual description of that function is printed. If sx\_arg is a number or *b*, *c* or *d*, then that chapter or appendix is printed. If sx\_arg is *tc*, then a table of contents is printed.



```
(if 'g_a 'g_b)
(if 'g_a 'g_b 'g_c ...)
(if 'g_a then 'g_b [...] [elseif 'g_c then 'g_d ...] [else 'g_e [...]])
(if 'g_a then 'g_b [...] [elseif 'g_c thenret] [else 'g_d [...]])
```

■ NOTE: The various forms of **if** are intended to be easily readable conditional statements—to be used in place of **cond**. There are two varieties of **if**: with and without keywords. The keyword-less variety is inherited from common MacLISP usage. A keyword-less, two argument **if** is equivalent to a one-clause **cond**, i.e., (**cond** (a b)). Any other keyword-less **if** must have at least three arguments. The first two arguments are the first clause of the equivalent **cond**, and all remaining arguments are shoved into a second clause beginning with *t*. Thus, the second form of **if** is equivalent to

```
(cond (a b) (t c ...)).
```

□ The keyword variety has the following grouping of arguments: a predicate, a then-clause, and an optional else-clause. The predicate is evaluated, and if the result is non-*nil*, the then-clause is performed, in the sense described later. Otherwise, that is, the result of the predicate evaluation was precisely *nil*, the else-clause is performed.

□ Then-clauses either consist entirely of the single keyword *thenret*, or start with the keyword *then*, and followed by at least one general expression. (These general expressions must not be one of the keywords.) To actuate a *thenret* means to cease further evaluation of the **if** and to return the value of the predicate just calculated. The performance of the longer clause means to evaluate each general expression in turn and then return the last value calculated.

□ The else-clause may begin with the keyword *else* and be followed by at least one general expression. The rendition of this clause is just like that of a then-clause. An else-clause may begin alternatively with the keyword *elseif* and be followed (recursively) by a predicate, then-clause, and optional else-clause. Evaluation of this clause, is just evaluation of an **if**-form, with the same predicate, then- and else-clauses.

```
(let l_args g_exp1 ... g_exprn)
```

■ RETURNS the result of evaluating *g\_exprn* within the bindings given by *l\_args*.

■ NOTE: *l\_args* is either *nil* (in which case **let** is just like **progn**) or it is a list of binding objects. A binding object is a list (symbol expression). When a **let** is entered, all of the expressions are evaluated and then simultaneously *lambda*-bound to the corresponding symbols. In effect, a **let** expression is just like a *lambda* expression except that the symbols and their initial values are next to each other, making the expression easier to understand. There are some added features to the **let** expression: A binding object can just be a symbol, in which case the expression corresponding to that symbol is *nil*. If a binding object is a list and the first element of that list is another list, then that list is assumed to be a binding template and **let** does a **desetaq**



on it.

(let\* l\_args g\_exp1 ... g\_expn)

- RETURNS the result of evaluating g\_expn within the bindings given by l\_args.
- NOTE: This is identical to **let** except the expressions in the binding list l\_args are evaluated and bound sequentially instead of in parallel.

(lexpr-funcall 'g\_function ['g\_arg1 ...] 'l\_argn)

- NOTE: This function has been obsoleted by **apply**, and is present for compatibility reasons only.

(listify 'x\_count)

- RETURNS a list of x\_count of the arguments to the current function (which must be a *lexpr*).
- NOTE: Normally arguments 1 through x\_count are returned. If x\_count is negative then a list of the last abs(x\_count) arguments are returned.

(map 'u\_func 'l\_arg1 ...)

- RETURNS l\_arg1
- NOTE: The function u\_func is applied to successive sublists of the l\_argi. All sublists should have the same length.

(mapc 'u\_func 'l\_arg1 ...)

- RETURNS l\_arg1.
- NOTE: The function u\_func is applied to successive elements of the argument lists. All of the lists should have the same length.

(mapcan 'u\_func 'l\_arg1 ...)

- RETURNS **nconc** applied to the results of the functional evaluations.
- NOTE: The function u\_func is applied to successive elements of the argument lists. All sublists should have the same length.

(mapcar 'u\_func 'l\_arg1 ...)

- RETURNS a list of the values returned from the functional application.
- NOTE: The function u\_func is applied to successive elements of the argument lists. All sublists should have the same length.

(mapcon 'u\_func 'l\_arg1 ...)

- RETURNS **nconc** applied to the results of the functional evaluation.



- NOTE: The function `u_func` is applied to successive sublists of the argument lists. All sublists should have the same length.

(`maplist 'u_func 'l_arg1 ...`)

- RETURNS a list of the results of the functional evaluations.
- NOTE: The function `u_func` is applied to successive sublists of the arguments lists. All sublists should have the same length.

You may find the following summary table useful in remembering the differences between the six mapping functions:

Arguments are	Value returned is		
	<code>l_arg1</code>	list of results	<code>nconc</code> of results
elements of list	<code>mapc</code>	<code>mapcar</code>	<code>mapcan</code>
sublists	<code>map</code>	<code>maplist</code>	<code>mapcon</code>

(`mfunction t_entry 's_disc`)

- RETURNS a LISP object of type binary composed of `t_entry` and `s_disc`.
- NOTE: `t_entry` is a pointer to the machine code for a function, and `s_disc` is the discipline (e.g., *lambda*).

(`oblist`)

- RETURNS a list of every interned symbol in the current LISP environment.
- NOTE: The name of this function is historical, and comes from *object list*. It used to be the case that all symbols lived on the same entity called the *oblist* or *obarray*.

(`or [g_arg1 ... ]`)

- RETURNS the value of the first non-*nil* argument or *nil* if all arguments evaluate to *nil*.
- NOTE: Evaluation proceeds left to right and stops as soon as one of the arguments evaluates to a non-*nil* value.

(`pop 'l_stack [g_into]`)

- RETURNS the top element on the stack `l_stack`. If `g_into` is given, a `setf` of `g_into` is done with the top element as the value. See `push` below.
- SIDE EFFECT: `l_stack` is modified to no longer include the top element.



(prog l\_vrbls g\_exp1 ...)

- RETURNS the value explicitly given in a return form or else *nil* if no return is done by the time the last *g\_exp1* is evaluated.
- NOTE: The local variables are *lambda*-bound to *nil*, then the *g\_exp1* are evaluated from left to right. This is a **prog** body (obviously) and this means that any symbols seen are not evaluated, but are treated as labels. This also means that **returns** and **gos** are allowed.

(prog1 'g\_exp1 ['g\_exp2 ...])

- RETURNS *g\_exp1*

(prog2 'g\_exp1 'g\_exp2 ['g\_exp3 ...])

- RETURNS *g\_exp2*
- NOTE: The forms are evaluated from left to right and the value of *g\_exp2* is returned.

(progn 'g\_exp1 ['g\_exp2 ...])

- RETURNS the last *g\_exp1*.

(progv 'l\_locv 'l\_initv g\_exp1 ...)

- WHERE *l\_locv* is a list of symbols and *l\_initv* is a list of expressions.
- RETURNS the value of the last *g\_exp1* evaluated.
- NOTE: The expressions in *l\_initv* are evaluated from left to right and then *lambda*-bound to the symbols in *l\_locv*. If there are too few expressions in *l\_initv*, then the missing values are assumed to be *nil*. If there are too many expressions in *l\_initv*, then the extra ones are ignored (although they are evaluated). Then the *g\_exp1* are evaluated left to right. The body of a **progv** is like the body of a **progn**, it is not a **prog** body. (Cf. **let**.)

(purcopy 'g\_exp)

- RETURNS a copy of *g\_exp* with new pure cells allocated wherever possible.
- NOTE: Pure space is never swept up by the garbage collector, so this should only be done on expressions that are not likely to become garbage in the future. In certain cases, data objects in pure space become read-only after a **dumplisp**, and then an attempt to modify the object results in an illegal memory reference.

(purep 'g\_exp)

- RETURNS *t* iff the object *g\_exp* is in pure space.



(push 'g\_element 'l\_stack)

(pushnew 'g\_element 'l\_stack)

- RETURNS modified l\_stack.
- NOTE: l\_stack is a fifo stack, and **push** conses g\_element into l\_stack. **pushnew** only pushes the element if it is not already there. See **pop** above.

(putd 's\_name 'u\_func)

- RETURNS u\_func
- SIDE EFFECT: This sets the function binding of symbol s\_name to u\_func.

(return ['g\_val])

- RETURNS g\_val (or *nil* if g\_val is not present) from the enclosing **prog** or **do** body.
- NOTE: This form is only valid in the context of a **prog** or **do** body.

(selectq 'g\_key-form [l\_clause1 ...])

- NOTE: This function is just like **caseq** (see earlier), except that the symbol *otherwise* also has the same semantics as the symbol *t* (signaling the default clause), when used as a comparator.

(setarg 'x\_argnum 'g\_val)

- WHERE x\_argnum is greater than zero and less than or equal to the number of arguments to the *lexpr*.
- RETURNS g\_val
- SIDE EFFECT: The *lexpr*'s x\_argnumth argument is set to g\_val.
- NOTE: This can only be used within the body of a *lexpr*.

(throw 'g\_val [s\_tag])

- WHERE if s\_tag is not given, it is assumed to be *nil*.
- RETURNS the value of (\*throw 's\_tag 'g\_val).

(\*throw 's\_tag 'g\_val)

- RETURNS g\_val from the first enclosing catch with the tag s\_tag or with no tag at all. Thus the value is accompanied by a change in control.
- NOTE: This is used in conjunction with **\*catch** to cause a clean jump to an enclosing context.

(unwind-protect g\_protected [g\_cleanup1 ...])

- RETURNS the result of evaluating g\_protected.



■ NOTE: Normally `g_protected` is evaluated and its value remembered, then the `g_cleanup` are evaluated, and, finally, the saved value of `g_protected` is returned. If something should happen when evaluating `g_protected` which causes control to pass through `g_protected`, and, thus, through the call to the `unwind-protect`, then the `g_cleanup` are still evaluated. This is useful if `g_protected` does something sensitive which must be cleaned up whether or not `g_protected` completes itself. Programs which 'temporarily' mess up a structure and then straighten the structure can use this scheme to protect the straightening-up process from being cut off by a keyboard interrupt.

### 4.3 Multiple value returns

Sometimes a function logically needs to return more than one value, but in most cases only one of them is used and it is therefore considered uneconomical to "cons-up" a structure for the unusual case. A function performing a division might return both the quotient and, as a second part, a remainder. A neat way to do this is by using multiple value returns. Only those functions which expect multiple values can receive them, and thus the default is to return a single value. The mechanism for using multiple values is explained below.

There are special functions which must be used to produce and receive multiple value. If a called function produces multiple values and the calling function does not request them, then all but the first value are discarded. If no values are produced, then the caller receives *nil* for a value. The maximum number of multiple values which can be returned by a functions is bound to the global variable *multiple-values-limit*.

The multiple value facility is completely compatible with Common LISP. Here are the functions to produce and receive multiple values:

`(values ['g_arg1 ... 'g_argn])`

■ RETURNS `g_arg1`, or *nil* if given no arguments. The `g_argi` are returned as multiple values. With the exception of the first, they can be accessed only by using one of the special forms below to receive them.

`(values-list 'l_arg)`

■ RETURNS the car of `l_arg`; but if received appropriately will exhibit the elements in the list `l_arg` as multiple values. This form is equivalent to `(apply 'values 'l_arg)`.

■ EXAMPLE: `(values-list '(1 2 3))` is equivalent to `(values 1 2 3)`.

`(multiple-value-call 'u_fun 'g_form1 ['g_form2 ...])`

■ RETURNS the result of calling `u_fun` with the results of all `g_formi` as arguments.



**(multiple-value-list 'g\_form)**

- RETURNS a list of the multiple values returned by *g\_form*. This form is equivalent to (multiple-value-call #'list 'g\_form).

**(multiple-value-prog1 'g\_form1 [ 'g\_form2 ... ])**

- RETURNS the values produced by *g\_form1*, after evaluating all *g\_formi*.

**(multiple-value-setq l\_varlist 'g\_form)**

- RETURNS the first value returned by *g\_form* after setting each variable in *l\_varlist* to the corresponding value returned by *g\_form* (the first variable gets the first value, and so on).
- NOTE: If there are more variables than returned values, then the remaining variables are given the value *nil*.

**(multiple-value-bind 'l\_varlist 'g\_values-form 'g\_form1 [ 'g\_form2 ... ])**

- RETURNS the result of evaluating *g\_formi*. The variables in *l\_varlist* are bound to the values returned by *g\_values-form*, and then all the *g\_formi* are evaluated.



1. The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that proper record-keeping is essential for the integrity of the financial system and for the ability to detect and prevent fraud.

2. The second part of the document outlines the specific procedures for recording transactions. It details the steps involved in the accounting cycle, from identifying the transaction to posting it to the appropriate ledger account.

3. The third part of the document discusses the role of internal controls in ensuring the accuracy of financial records. It describes various control measures, such as segregation of duties and independent verification, that are designed to minimize the risk of errors and fraud.

4. The fourth part of the document addresses the importance of regular audits in the financial reporting process. It explains how audits provide an independent assessment of the reliability of the financial statements and help to identify areas for improvement.

5. The fifth part of the document discusses the impact of technology on financial reporting. It highlights the benefits of using accounting software and other digital tools to streamline the reporting process and improve the accuracy of the data.

## 5 Input/output

- 5.1 Introduction 5-1
- 5.2 Functions 5-2
- 5.3 Format 5-11
  - 5.3.1 Format directives 5-13
  - 5.3.2 Format argument processing 5-17
  - 5.3.3 Conditional and iterative directives 5-18
    - 5.3.3.1 Alternate strings 5-18
    - 5.3.3.2 Iteration strings 5-19





## 5 Input/output

### 5.1 Introduction

The following functions are used to read from and write to external devices (e.g. files) and programs through pipes. All I/O goes through the LISP data type called the **port**. A port may be open for either reading or writing but usually not both simultaneously (see **fileopen**). There are only a limited number of ports (at least 20) and they are not reclaimed unless they are **closed**. All ports are reclaimed by a **resetio** call, but this drastic action is not necessary if the program closes ports that it uses.

If a port argument is not supplied to a function that requires one, or if a bad port argument (such as *nil*) is given, then FRANZ LISP uses the default port according to this scheme: if input is being done, then the default port is the value of the symbol *piport* and, if output is being done, then the default port is the value of the symbol *poport*. Furthermore, if the value of *piport* or *poport* is not a valid port, then the standard input or standard output is used, respectively.

The standard input and standard output are usually the keyboard and terminal display unless your job is running in the background and its input or output is connected to a pipe. All output that goes to the standard output also goes to the port *ptport*, if it is a valid port. Output destined for the standard output does not reach the standard output if the symbol *^w* is non-*nil*, although it still goes to *ptport* if *ptport* is a valid port.

FRANZ LISP has borrowed a convenient shorthand notation from the Unix operating system 'C' shell concerning naming files. If a file name begins with *~* (tilde), and the symbol *tilde-expansion* is bound to something other than *nil*, then FRANZ LISP expands the file name. It takes the string of characters between the leading tilde, and the first slash as a user-name. Then, that initial segment of the filename is replaced by the home directory of the user. The null username is taken to be the current user.

On Unix systems, having gone to the effort of searching the password file, FRANZ LISP remembers the user directory, in case it gets asked to do so again. Tilde-expansion is performed in most places in which file names are expected, and in particular, the following functions: **cfasl**, **chdir**, **fasl**, **ffasl**, **fileopen**, **infile**, **load**, **outfile**, **probe**, **sys:access**, **sys:unlink**.

The programmer should be careful to note which of the functions reference file names and which refer to ports.



## 5.2 Functions

(**cfasl** 'st\_file 'st\_entry 'st\_funcname ['st\_disc ['st\_library]])

■ RETURNS *t*.

■ SIDE EFFECT: This is used to load in a foreign function (see chapter 18). The object file *st\_file* is loaded into the LISP system. *St\_entry* should be an entry point in the file just loaded. The function binding of the symbol *s\_funcname* is set to point to *st\_entry* so that, when the LISP function *s\_funcname* is called, *st\_entry* is run. *st\_disc* is the discipline to be given to *s\_funcname*. *st\_disc* defaults to "subroutine" if it is not given or if it is given as *nil*. If *st\_library* is non-*nil*, then after *st\_file* is loaded, the libraries given in *st\_library* are searched to resolve external references. The form of *st\_library* should be something like "-llibname". The C library ("-lc") is always searched so that when loading in a C file, you probably will not need to specify a library.

■ NOTE: This function may be used to load the output of the assembler, C compiler, Fortran compiler, and Pascal compiler, but NOT the LISP compiler. Use **fasl** for that. If a file has more than one entry point, then use **getaddress** to locate and setup other foreign functions.

□ It is an error to load in a file that has a global entry point of the same name as a global entry point in the running LISP. As soon as you load in a file with **cfasl**, its global entry points become part of the LISP's entry points. Thus, you cannot **cfasl** in the same file twice unless you use **removeaddress** to change certain global entry points to local entry points.

(**charent** 'p\_port)

■ RETURNS the number of characters left on the current line in *p\_port*.

(**close** 'p\_port)

■ RETURNS *t*.

■ SIDE EFFECT: The specified port is drained and closed, releasing the port.

■ NOTE: The standard defaults are not used in this case since you probably never want to close the standard output or standard input.

(**cprintf** 'st\_format 'xfst\_val ['p\_port])

■ RETURNS *xfst\_val*.

■ SIDE EFFECT: The operating system formatted output function **printf** is called with arguments *st\_format* and *xfst\_val*. If *xfst\_val* is a symbol, then its print name is passed to **printf**. The format string may contain characters that are printed literally, and it may contain special formatting commands preceded by a percent sign. The complete set of formatting characters is described in the operating system manual. Some useful ones are "%d" for printing a *fixnum* in decimal, "%f" or "%e" for printing a *flonum*, and "%s" for printing a character string (or print name of a symbol).



■ **EXAMPLE:** (`cprintf "Pi equals %f" 3.14159`) prints *Pi equals 3.14159*.

(`drain ['p_port]`)

■ **RETURNS** *nil*.

■ **SIDE EFFECT:** If this is an output port, then the characters in the output buffer are all sent to the device. If this is an input port, then all pending characters are flushed. The default port for this function is the default output port.

(`fasl 'st_name ['st_mapf ['g_warn]]`)

■ **WHERE** `st_mapf` and `g_warn` default to *nil*.

■ **RETURNS** *t* if the function succeeded, *nil* otherwise.

■ **SIDE EFFECT:** This function is designed to load in an object file generated by the LISP compiler *Liszt*. File names for object files usually end in '.o,' so **fasl** append '.o' to `st_name`, if it is not already present. If `st_mapf` is non-*nil*, then it is the name of the map file to create. **fasl** writes in the map file the names and addresses of the functions it loads and defines. Normally, the map file is created (i.e. truncated if it exists), but if (`sstatus appendmap t`) is done, then the map file is appended. If `g_warn` is non-*nil* and if a function is loaded from the file that is already defined, then a warning message is printed.

■ **NOTE:** **fasl** only looks in the current directory for the file to load. The function **load** looks through a user-supplied search path and calls **fasl** if it finds a file with the same root name and a .o extension. In most cases, you should use the function **load** rather than calling **fasl** directly.

(`fileopen 's_filename 's_mode`)

■ **RETURNS** a port for reading or writing based on the file `s_filename`. The `s_mode` is one of *r*, *w*, or *a*, (for read, write, append), or *r+*, *w+*, or *a+*, each of which permits both reading and writing on a port provided that **fseek** is done between changes in "direction." In the case of writing, the file will be created if it does not already exist. The directory search-path for file name resolution is **not** used.

(`filepos 'p_port ['x_pos]`)

■ **RETURNS** the current position in the file if `x_pos` is not given or else `x_pos` if `x_pos` is given.

■ **SIDE EFFECT:** If `x_pos` is given, the next byte to be read or written to the port is at position `x_pos`.

(`filestat 'st_filename`)

■ **RETURNS** a vector containing various numbers that the operating system assigns to files. If the file does not exist, an error is invoked. Use **probeif** to determine if the file exists.



■ NOTE: The individual entries can be accessed by mnemonic functions of the form **filestat-field**, where field may be any of: **dev**, **ino**, **mode**, **mtime**, **atime**, **ctime**, **gid**, **rdev**, **nlink**, **size**, **type**, or **uid**. These functions are described below.

```
(filestat-atime 'v_filestat)
(filestat-ctime 'v_filestat)
(filestat-dev 'v_filestat)
(filestat-gid 'v_filestat)
(filestat-ino 'v_filestat)
(filestat-mode 'v_filestat)
(filestat-mtime 'v_filestat)
(filestat-nlink 'v_filestat)
(filestat-rdev 'v_filestat)
(filestat-size 'v_filestat)
(filestat-type 'v_filestat)
(filestat-uid 'v_filestat)
```

■ RETURNS *fixnums* that represent information about a file whose status information (returned by **filestat** above) is in the vector **v\_filestat**. These numbers are operating-system dependent and not all functions may have meaning in all operating systems. When the function has no meaning, it returns *nil*. These functions return, respectively, the last time the file was accessed (as an integer in the basic unit of time in the native operating system), the last time the file changed state (again in basic time units), the number of the device on which the file resides, the group identification number of the owner of the file, the number of the file (the i-node number in Unix), the protection mode of the file, the last time the file was modified (in basic time units), the number of 'hard links' to the file (meaningful in Unix only), the type of device if the file is a device (probably meaningful in Unix only), the size of the file (in basic units of the operating system, e.g. bytes in Unix), the type of the file, and the user identification number of the owner of the file.

```
(flatc 'g_form ['x_max])
```

■ RETURNS the number of characters required to print **g\_form** using **patom**. If **x\_max** is given and, if **flatc** determines that it returns a value greater than **x\_max**, then it gives up and returns the current value it has computed. This is useful if you just want to see if an expression is larger than a certain size.

```
(flatsize 'g_form ['x_max])
```

■ RETURNS the number of characters required to print **g\_form** using **print**. The meaning of **x\_max** is the same as for **flatc**.

■ NOTE: Currently this just **explodes** **g\_form** and checks its length.



(fseek 'p\_port 'x\_offset 'x\_flag)

- RETURNS the position in the file after the function is performed.
- SIDE EFFECT: This function positions the read/write pointer before a certain byte in the file. If *x\_flag* is 0 then the pointer is set to *x\_offset* bytes from the beginning of the file. If *x\_flag* is 1 then the pointer is set to *x\_offset* bytes from the current location in the file. If *x\_flag* is 2 then the pointer is set to *x\_offset* bytes from the end of the file.

(infile 's\_filename)

- RETURNS a port ready to read *s\_filename*.
- SIDE EFFECT: This tries to open *s\_filename*, and, if it cannot or if there are no ports available, it gives an error message.
- NOTE: To allow your program to continue on a file-not-found error, you can use something like:

```
(cond ((null (setq myport
                  (car (errset (infile name) nil))))
      (patom "couldn't open the file")))
```

which sets *myport* to the port to read from if the file exists or prints a message if it could not open it and also sets *myport* to *nil*. To simply determine if a file exists, use *probef*.

(load 's\_filename ['st\_map ['g\_warn]])

- RETURNS *t*.
- NOTE: The function of **load** has changed since previous releases of FRANZ LISP and the following description should be read carefully.
- SIDE EFFECT: **load** now serves the function of both **fasl** and the old **load**. **load** searches a user-defined search path for a LISP source or object file with the filename *s\_filename* (with the extension appropriate). The search path that **load** uses is the value of (status load-search-path). The default is (|.| /usr/lib/lisp), which means: look in the current directory first and then */usr/lib/lisp*. The file that **load** looks for depends on the last two characters of *s\_filename*. If *s\_filename* ends with *.l*, then **load** only looks for a file name *s\_filename* and assumes that this is a FRANZ LISP source file. If *s\_filename* ends with *.o*, then **load** only looks for a file named *s\_filename* and assumes that this is a FRANZ LISP object file to be **fasl**ed in. Otherwise, **load** first looks for *s\_filename.o*, then *s\_filename.l*, and, finally, *s\_filename* itself. If it finds *s\_filename.o*, it assumes that this is an object file; otherwise, it assumes that it is a source file. An object file is loaded using **fasl** and a source file is loaded by reading and evaluating each form in the file. The optional arguments *st\_map* and *g\_warn* are passed to **fasl** should **fasl** be called.
- NOTE: **load** requires a port to open the file *s\_filename*. It then *lambda* binds the symbol *piport* to this port and reads and evaluates the forms.



**(makereadtable ['s\_flag])**

- WHERE if `s_flag` is not present it is assumed to be *nil*.
- RETURNS a readtable equal to the original readtable if `s_flag` is non-*nil*, or else equal to the current readtable. See Chapter 7 for a description of readtables and their uses.

**(msg [l\_option ...] ['g\_msg ...])**

- NOTE: This function is intended for printing short messages. Any of the arguments or options presented can be used any number of times in any order. The messages themselves (`g_msg`) are evaluated, and then they are transmitted to **patom**. Typically, they are strings, which evaluate to themselves. The options are interpreted specially:

msg option summary	
(P p_portname)	Causes subsequent output to go to the port <code>p_portname</code> which should have been opened previously.
B	Print a single blank.
(B 'n_b)	Evaluate <code>n_b</code> and print that many blanks.
N	Print a single newline by calling <b>terpr</b> .
(N 'n_n)	Evaluate <code>n_n</code> and transmit that many newlines to the stream.
D	Drain the current port.

**(nwritn ['p\_port])**

- RETURNS the number of characters in the buffer of the given port but not yet written out to the file or device. The buffer is flushed automatically when filled or when **terpr** is called.

**(outfile 's\_filename ['st\_type])**

- RETURNS a port or *nil*.
- SIDE EFFECT: This opens a port to write `s_filename`. If `st_type` is given and if it is a symbol or string whose name begins with *a*, then the file is opened in append mode; that is, the current contents are not lost, and the next data is written at the end of the file. Otherwise, the file opened is truncated by **outfile** if it existed beforehand. If there are no free ports, **outfile** returns *nil*. If one cannot write on `s_filename`, an error is signalled.



(**patom** 'g\_exp ['p\_port])

- RETURNS **g\_exp**.
- SIDE EFFECT: **g\_exp** is printed to the given port or the default port. If **g\_exp** is a symbol or string, the print name is printed without any escape characters around special characters in the print name. If **g\_exp** is a list, then **patom** has the same effect as **print**.

(**pntlen** 'xfs\_arg)

- RETURNS the number of characters needed to print **xfs\_arg**.

(**portp** 'g\_arg)

- RETURNS **t** iff **g\_arg** is a port.

(**pp** [l\_option] s\_name1 ...)

- RETURNS **t**.
- SIDE EFFECT: If **s\_name<sub>i</sub>** has a function binding, it is pretty-printed; otherwise, if **s\_name<sub>i</sub>** has a value, then that is pretty-printed. Normally, the output of the pretty-printer goes to the standard output port *poport*. The options allow you to redirect it.

pp option summary	
(F s_filename)	Direct future printing to s_filename.
(P p_portname)	Causes output to go to the port p_portname, which should have been opened previously.
(E g_expression)	Evaluate g_expression and do not print.

(**princ** 'g\_arg ['p\_port])

- EQUIVALENT TO: **patom**.

(**print** 'g\_arg ['p\_port])

- RETURNS *nil*.
- SIDE EFFECT: Prints **g\_arg** on the port **p\_port** or the default port. For objects that cannot be read back in, the convention is used that a sharp-sign (#) prefixes the data printed.
- EXAMPLE: The LISP objects *other*, *bcd*, *vector*, *hash-table*, *package*, *array*, and *port* now print as #<...>.. Reading the value of printing one of the above objects is an error.



---

```
=> *package*
#<"user" package>
=> (getd 'car)
#<bcd 0x16B2E lambda>
=> (make-hash-table)
#<eq hash-table with 0 entries and 67 buckets @ #xe45cc>
=> Standard-Input
#<port stdin>
=> (vector 10)
#<vector 1>
=> (marray nil nil nil 10 nil)
#<array 10>
=> (getd 'format)
#<bcd 0xD4800 lambda>
=> (fake #xd4800)
#<other 0xD4800>
; Known to be compiled code, which is type other
```

---

(probe 'st\_file)

- RETURNS *t* iff the file *st\_file* exists.
- NOTE: Just because it exists does not mean you can read it.

(pp-form 'g\_form ['p\_port])

- RETURNS *t*
- SIDE EFFECT: *g\_form* is pretty-printed to the port *p\_port* (or *poport* if *p\_port* is not given). This is the function that **pp** uses. **pp-form** does not look for function definitions or values of variables, it just prints out the form it is given.
- NOTE: This is useful as a top-level printer (See Chapter 13).

(ratom ['p\_port ['g\_eof]])

- RETURNS the next atom read from the given or default port. On end of file, *g\_eof* (default *nil*) is returned.

(read ['p\_port ['g\_eof]])

- RETURNS the next LISP expression read from the given or default port. On end of file, *g\_eof* (default *nil*) is returned.
- NOTE: An error occurs if the reader is given an ill-formed expression. The most common error is too many right parentheses. (Note that this is not considered an error in MacLISP).



**(readc ['p\_port ['g\_eof]])**

- RETURNS the next character read from the given or default port. On end of file, *g\_eof* (default *nil*) is returned.

**(readline ['p\_port])**

- RETURNS a string containing all characters in the stream *p\_port* up to, but not including, the next newline. The null string is returned if the first character is a newline.

**(readdir ['t\_dirname])**

- RETURNS a list of strings, one for each file in the named directory.
- NOTE: *t\_dirname* is a string representing the name of a directory. It may be *..*, meaning return a list of the current working directory. (This is the default if the argument is omitted.) All files are listed except *."* and *.."*. No tilde-expansion is done on the argument.

**(readlist 'l\_arg)**

- RETURNS the LISP expression read from the list of characters in *l\_arg*.

**(removeaddress 's\_name1 ['s\_name2 ...])**

- RETURNS *nil*
- SIDE EFFECT: The entries for the *s\_namei* in the LISP symbol table are removed. This is useful if you wish to *cfasl* in a file twice, since it is illegal for a symbol in the file you are loading to already exist in the LISP symbol table.

**(resetio)**

- RETURNS *nil*
- SIDE EFFECT: All ports except the standard input, output, and error are closed.

**(sload 's\_file)**

- SIDE EFFECT: The file *s\_file* (in the current directory) is opened for reading, and each form is read, printed, and evaluated. If the form is recognizable as a function definition, only its name is printed; otherwise, the whole form is printed.
- NOTE: This function is useful when a file refuses to load because of a syntax error and you would like to determine where the error is.

**(sprintf 't\_control ['arg1 ...])**

- RETURNS the formatted string corresponding to the returned value of the C library *sprintf()* function.



- NOTE: The (backslash) escapes do **not** work. `t_control` must be a string.

(tab 'x\_col ['p\_port])

- SIDE EFFECT: Enough spaces are printed to put the cursor on column `x_col`. If the cursor is beyond `x_col` to start with, a **terpr** is done first.

(terpri ['p\_port])

(terpr ['p\_port])

- RETURNS nil
- SIDE EFFECT: A terminate line character sequence is sent to the given port or the default port. This also drains the port.

(tilde-expand 'st\_name)

- RETURNS `st_name` with all `~`'s expanded with absolute pathnames in their place.
- NOTE: this function is not available in all versions of FRANZ LISP.

(truename 'p\_port)

- RETURNS the name of the file to which `p_port` refers.

(tyi ['p\_port])

- RETURNS the *fixnum* representation of the next character read. On end of file, `-1` is returned.

(tyipeek ['p\_port])

- RETURNS the *fixnum* representation of the next character to be read.
- NOTE: This does not cause an official 'read' of the character, it just peeks at it and returns the value which would be returned if it were read.

(tyo 'x\_char ['p\_port])

- RETURNS `x_char`.
- SIDE EFFECT: The character whose *fixnum* representation is `x_code` is printed as a character on the given output port or the default output port.

(untyi 'x\_char ['p\_port])

- SIDE EFFECT: `x_char` is put back in the input buffer so a subsequent **tyi** or **read** reads it first.
- NOTE: A maximum of one character may be put back.



(**wide-print-list** 'g\_exp [keyword value])

■ RETURNS *nil*.

■ SIDE EFFECT: *g\_exp* is print to *oport*, or the port specified by the keyword argument, *:port*. **wide-print-list** prints as many entities on one line as will fit, assuring that no LISP object other than a list is broken across a line boundary. In particular, atoms and numbers remain on one line.

*:port* p\_port

■ p\_port is used at the output port.

*:left-margin* x\_lmar

■ The left margin, if other than 0.

(**y-or-n-p** ['t\_message])

■ RETURNS *t* if an answer of "y" is read from the user, *nil* otherwise. If *t\_message* is given, then prompt the user with this string before reading.

■ NOTE: *piport* and *oport* are used as the querying ports.

(**zapline**)

■ RETURNS *nil*

■ SIDE EFFECT: All characters up to and including the line termination character are read and discarded from the last port used for input.

■ NOTE: This is used as the *macro* function for the semicolon character when it acts as a comment character.

### 5.3 Format

**Format** is a output formatter somewhat in the style of the **fprintf()** 'C' run-time library program. The arguments to **format** include an output port, a control string, and the values to be printed. The values passed to **format** are printed according to the directives in the control string.

In the simplest case, the **format** control string has no directives and is passed no variables. In this case, **format** just prints its second argument.

(**format** t "Hello world") = Hello worldnil

As we see from the previous example, **format** prints some expression, and then returns a value *nil*. If, instead of *t* (which indicates the standard output), we had used *nil* as the value for the port, **format** would have printed nothing, but returned a string, "Hello world".



(format 'p\_port 's\_ctrl ['g\_arg ...])

■ RETURNS *nil* if *p\_port* is non-*nil* otherwise returns a string determined by *s\_ctrl* and the arguments.

■ WHERE *p\_port* is one of: *t* (the user's terminal), an output port opened with **out-file** (or **fileopen**), or *nil* (which causes **format** to return a string). *s\_ctrl* is a control string, which may also contain *directives*, prefaced by a tilde ("~"), as explained, below.

□ The number of arguments should correspond to the number of values expected by the control string. All arguments are processed according to the number of directives contained in the control string. Extra arguments are discarded. When an insufficient number of arguments are given, default values may be inserted.

■ SIDE EFFECT: when *p\_port* is non-*nil*, prints a string to *p\_port*.

□ In the example, above, the control string was a simple string, containing no directives and there were no arguments to the string. The remainder of this section will deal with more complex control strings which contain directives which are used to manipulate the output.

□ Directives are embedded in the control string and always begin with the *tilde* "~" character. The tilde may be followed by optional **prefixes** and **modifiers**. The

---

In this section on the **format** function, strings are represented as "Hello". Non-literal components of the string are shown in bold face, as in "The number **~m,nd** is even.~%", where **m,n** are parameters to the "~d" directive. A possible literal string corresponding to the previous string would be "The number 5,0d is even.~%". The brackets indicate that the parameters to the directive are optional.

---

directive is terminated by a single character that determines the nature of the directive. (In the case of alphabetic characters, the case is immaterial.) In fact directives are often referred to by the terminating character of the directive. The prefixes are numeric or symbolic constants separated by commas. These prefixes parameterize the directive and are used to control aspects of the output or side-effects produced by the directive. Modifiers are one or more of the special characters at-sign "@" or colon ":". Each directive usually consumes one argument to **format**, but complex directives may iterate or otherwise consume more than one argument. Further, some directives exist entirely for their side-effects, which may alter the order of processing of the arguments to **format**. Simple directives appear alone, but complex directives for iteration will appear together and an entire 'expression' may involve a string of directives delimited by 'opening' and 'closing' directives. All of these variations will be described and illustrated in the following text. In general, the form of a directive is:

"[<prefix>[,<prefix>...][<modifier>...]<character>"

When more than one prefix appears, they are separated by commas. Multiple modifiers are juxtaposed without commas. For example, the directive "~d" causes the argument to be printed in decimal format. However, "~nd" causes the argument to be



printed to *n* decimal places (using spaces as pad characters), while "*n,md*" will cause the integer to be printed to *n* decimal places using the character whose decimal representation is *m* as the pad character. Prefixes are always optional, and null prefixes may be indicated by just a comma.

---

```

; The ~% outputs a carriage return.
=> (format t "Today's daily number is: ~d ~%" 1024)
Today's daily number is: 1024
nil
; The number will be printed to four places but
; the pad character will be a space.
=> (format t "Today's daily number is: ~4d ~%" 304)
Today's daily number is: 304
nil
; 48 is the decimal representation of a zero.
=> (format t "Today's daily number is: ~4,48d ~%" 304)
Today's daily number is: 0304
nil
=>

```

---

The prefix parameter will sometimes indicate a character to be printed, such as the padding character in the case of "*d*" above. Since **format** expects a decimal number, ASCII characters should either be specified by their decimal values (e.g., 48 = "0"), or by using a quote sign (e.g., "*4,0d*"). In addition, the prefix character can be the letter "*v*" or the sharp-sign character, "#". The "*v*" prefix directs **format** to use the value of the current argument as the prefix to the directive. This syntax is illustrated in description of the "*q*", directive. The "#" prefix substitutes the number of remaining arguments for the prefix character. The use of the sharp-sign is illustrated in a number of examples below.

### 5.3.1 Format directives

The following is a description of the **format** directives. These have been modeled after those in **MacLISP** and **ZetaLISP**. The symbols *arg* and *argi* will sometimes be used to indicate the corresponding argument or arguments to the directive. The symbol *argi* indicates the *i*th argument to the directive.

#### "~width"

■ Prints any argument as it would be printed by **princ** (without escapes). The argument is left-justified to width *width* with spaces. This directive may be used to print symbols or lists.



**"~widths"**

■ This directive is identical to **"~a"** except that the print function is **prin1**. Note that symbols print differently when format is outputting to a port rather than returning a string. When format outputs to a port (the **p\_port** argument is **t** or a port), symbol names are printed with vertical bars when they contain characters that are not in the *ccharacter* class, e.g. spaces. When **format** returns a string (the **p\_port** argument is **nil**), symbols names are returned with special characters 'escaped' by a backslash (**'\'**).

**"~mincolw,padchard"**

■ This directive prints the decimal representation of its argument, without a decimal point. If the argument is not a number, it is printed in **"~a"** format in decimal radix. The argument is left-justified in a column of width *mincolw*; if the number will not fit in this width, as many columns as required are printed. *padchar* can be used to specify a padding character in decimal (default is a space but a useful value is decimal 48 which is an ASCII **'0'**).

**"~mincolw,padcharo"**

■ This directive is identical to **"~d"** except that it prints the argument in octal radix. Note that the argument must be a number. See also **"~r"**.

**"~digitsf"**

■ **"~f"** prints the argument in floating-point format. The argument is rounded to a precision of *digits* digits. The value of *digits* must be at least two since a decimal point is always printed. The decimal pointed is floated to be consistent with standard scientific notation so that extremely large numbers and extremely small number are printed in exponential notation. If the argument is not a number it is printed in **"~a"** format. Unlike the previous directives, the *digits* prefix is not the minimum column width; rather, it is total number of digits of precision desired.

**"~digitse"**

■ This directive prints its argument in exponential notation. The prefix means the same thing to this directive as it does to the **"~f"** directive above.

**"~radix,mincolw,padchar:@r"**

■ The **"~r"** is a general purpose number formatter. The *mincolw* and *padchar* prefixes have the same meaning to **"~r"** as they do to **"~d"**. The *radix* prefix causes the argument to be printed in that radix, e.g. **"~10r"** is equivalent to **"~d"**. Nonnumeric arguments are printed using **"~a"**.

□ If no prefix is given, the numerical argument is printed as text, e.g. if the argument is 4, then **"~r"** prints **"four"** (cardinal value). The modifiers **:** and **@** affect the nature



of the text printed: `"~:r"` prints 4 as the ordinal string *"fourth"*, `"~@r"` prints the Roman numeral *"IV"*, and `"~:@r"` prints *"IIII"* (old-style Roman numeral).

---

```
=> (format t "The ~:r one on the right.~%" 1)
The first one on the right.
nil
; ~% prints a carriage return.
=> (format t "Total value of door number ~r: ~r dollars!~%" 3 12332)
Total value of door number three: twelve thousand three hundred
thirty-two dollars!
nil
; the next example illustrates the use of the ~* directive which
; allows the user to evaluate the arguments out of order.
=> (format t "The octal representation of ~d is ~:* ~8r.~%"
      18 22)
The octal representation of 18 is 22.
nil
```

---

### `"~:@c"`

■ `"~c"` prints the argument (which should be a *fixnum* representing an ASCII character) in a more human-readable form. `"~c"` prints the non-printable (control) characters represented by the *fixnum* argument in a slashified octal form, e.g. printing 3 as *"\003"*. `"~:c"` prints non-printable characters as an uppercase character prefixed by a caret, e.g. *"^C"*. Some characters such as decimal 32 (a space) are printed as their name, viz. *"space"*. The `"~@c"` form prints the character in LISP readable form, e.g. 3 as *"#^C"*. In all cases, the printable characters are printed in their keyboard form. p

---

The `"~c"` directive allows the character to be displayed along with any control bits that may be set. In ZetaLISP, an alternate character set may be used to indicate non-printing characters such as control characters (using the unmodified `"~c"` directive) while characters printed with the `"~:c"` directive will have the control bits spelled out. For example, a character may print out as *"Control-Meta-X"*. Mouse characters will print out *"Mouse-<button number>-<number of clicks>"*. This is not, yet, available in FRANZ LISP.

---

### `"~number%"`

■ The `"~%"` directive outputs a carriage-return. In most UNIX modes, the operating system will generate a line-feed automatically upon receipt of the carriage-return. A carriage-return may also be inserted directly into the control string, but this directive can be used to create more readable code. If it becomes necessary to insert a carriage-return, which should not be printed, into code to make it more readable, a simple tilde prefixing the carriage-return will 'escape' it. The prefix *number* specifies how many carriage-returns to print.



**"~numberx"**

- **"~numberx"** outputs *number* spaces.

**"~number&"**

- The **"~&"** directive calls for a 'fresh line', meaning that if the cursor is not in the first column of the current line, output a carriage return, otherwise do nothing. If the prefix is present, a fresh line is output followed by *number* - 1 carriage returns.

**"~number:/"**

- **"~/**" outputs a formfeed on a line oriented output. **"~number/**" outputs *number* formfeeds. **"~:/**" does the same thing as **"~/**" because FRANZ LISP does not support screen-oriented output (i.e. cursor-addressing and similar operations).

**"~number~"**

- **"~"** outputs a tilde. The prefix specifies how many to output.

**"~column,offset:t"**

- **"~t"** outputs two spaces.

---

In ZetaLISP this directive can be used in screen-oriented output to move to a particular column. Since screen-oriented operations are not supported by FRANZ LISP, this directive is somewhat limp.

---

**"~RETURN"**

**"~SPACE"**

- The tilde character can be used to 'escape' characters which appear in the control string but which should not be printed. Most commonly these characters consist are SPACE and RETURN. For example, in some circumstances the control string may stretch out over two or more lines, and the user may wish to insert carriage returns to make more readable code. This can be done if the carriage return is the character immediately following the tilde. Similarly, a single space immediately following a tilde will not be printed.

**"~:@p"**

- This directive modifies text to include plural syntax. It outputs a plural suffix based on the value of its argument. If the argument is not 1, then the character "s" is printed. **"~@p"** prints "ies" if the argument is not one, otherwise it prints "y". The colon modifier causes the *last* argument, rather than the *current* argument to be used.



---

```
=> (format t "Yes, we have ~a banana~p!~ ~%" "no" 2)
Yes, we have no bananas!
nil
; the "~[" directive is explained in the following section
=> (format t "Here ~[~1;is~::;are~] ~:*~r pupp~:@p.~%" 3)
Here are three puppies.
nil
```

---

### "~v:@q"

■ This directive provides a means of calling functions from within **format** control strings. The name of the function should be the argument. The called function can determine if the **:** or **@** modifiers were used by referencing the global variables **colon-flag** and **at-sign-flag**. Arguments may be passed to the called function with the **v** prefix character. The value returned by the function is ignored by **format** (i.e. it is not printed).

---

```
=> (defun test-~q-directive (arg)
      (princ "The value of at-sign-flag is: ")
      (princ at-sign-flag) (terpri)
      (princ "The value of colon-flag is: ")
      (princ colon-flag) (terpri)
      (princ "The value of arg is: ")
      (princ arg) (terpri))
test

=> (format t "~v@q" "dummy" 'test-~q-directive)
The value of at-sign-flag is: t
The value of colon-flag is: nil
The value of arg is: dummy
nil
```

---

## 5.3.2 Format argument processing

The following directives control the application of arguments to the control string. Normally, the arguments are applied in order, according to the requirements of the directives. There are, however, circumstances that require altering that order, and these directives provide a means of doing so. We have already seen in the last example a way of applying a single argument to more than one directive using the **"~\*"** directive.



"~number:~"

- The current argument is ignored. With a prefix, *number* arguments are ignored. With the : modifier, arguments are ignored *backwards* from the current argument. Thus "~:~" reprocessed the previous argument. The effect of this directive in iterative operations is described in the next section.

"~numberg"

- Causes the *number* argument to become the current argument. All other argument references will be relative to argument *number*. Arguments are *zero-indexed*, therefore "~0g" will go to the first argument. The use of this directive in iterative operations is detailed below.

### 5.3.3 Conditional and iterative directives

The following directives allow the user to specify alternate control strings that can be evaluated in a particular sequence or on a conditional basis. In general, the form of these directives is a block of text beginning with one of "~[", "~{", or "~<" and ending with a corresponding directive ("~]", "~}", or "~>"). *Clauses* or *items* in the block are demarcated by the separator directive "~;". In some instances a *default* control string can be included as the last element of the block and prefixed by a special form of the clause separator: "~:~". In most cases, if no default is specified and none of the conditions are satisfied, no alternative is printed.

#### 5.3.3.1 Alternate strings

"~:@["

"~]"

"~:~;"

"~[string1~;string2...~:~;stringn~]"

- The "~[" and "~]" directives delimit a zero-ordered list of alternate control strings that are separated by "~;" directives. The entire "~[...~]" directive expression takes a single argument, the *index* of a control string in the list. Only that control string will be processed by *format*—the other control strings in the directive expression are ignored. The first control string is selected by an argument of zero. Unless a default is given using the modified "~:~" separator directive as the last separator in the list, no control string is processed if the value for the argument is out of range.

□ The modified forms of the "~[" are described separately below.

"~:[["

"~:[false~;true~]"

- This directive will select the *false* control string if the argument to the "~:[[" directive evaluates to *nil*, otherwise it will select the *true* control string.



```
"~@["
"~@[true~]"
```

■ An alternative to the previous directive: the control string *true* is only used if the argument is non-*nil*. If the value of the argument is *nil*, it is discarded, the argument pointer is advanced to the next argument, and no control string is processed. If the argument evaluates to *t*, then the pointer is not advanced and the argument will be the current argument for the next directive.

```
"~tag1,tag2,...;"
"~begin1,end1,begin2,end2,...;"
"~:;"
"~["tag11,tag12,...;string1~begin21,end21,begin22,end22,...;string2~:;default"
```

This is the most complicated of the alternate-string directives.<sup>1</sup> The *tagij* strings are comma separated lists of values which may serve as a tag for the corresponding *stringi*. If the argument to the "[" directive is *eq* to one of the *tagij*, the corresponding *stringi* is processed. If a colon modifies the separator directive, the tags list should be a list of pairs of values, each pair representing a range of values for the argument. If the argument falls within the range (inclusive) of any of the pairs *beginij*, *endij*, the corresponding control string *stringi* is processed. For example, "["'0,9;;numeral~]" will use the string "numeral" if the value of the argument to this directive expression is between 0 and 9. If a separator directive appears without prefixes, the corresponding control string is processed as a default if the argument to the directive expression does not select any other alternative control string.

### 5.3.3.2 Iteration strings

In certain circumstances, *format* may be called repeatedly to process data that is in the same form, for example, the elements of an association list or property list. To illustrate, let us assume that we want a nice way of listing the elements of the property list of some symbol. We might try the following:

---

```
=> (putprop 'Clyde 'elephant 'type)
elephant
=> (putprop 'Clyde 'grey 'color)
grey
=> (putprop 'Clyde 'African 'subtype)
African
=> (plist 'Clyde)
(subtype African color grey type ...)
```

<sup>1</sup>Note, for example, that the syntax of the separator directive is slightly different from that in the previous cases. In the template, the *tagij* are actually prefixes to the separator directive, therefore the tilde precedes the *tagij*, which are followed by the (possibly modified) separator directive.



```
=> (format t "Symbol: ~a ~%-{ property: ~10s value: ~10s ~%-}" 'Clyde
(plist 'Clyde))
1 1.
Symbol: Clyde
  property: subtype  value: African
  property: color    value: grey
  property: type     value: elephant
nil
=>
```

---

```
"~:@{""
"~:}"
"~{string}"
"~:{string}"
"~@{string}"
"~:@{string}"
"~{string~}"
"~:{string~}"
"~@{string~}"
"~:@{string~}"
```

■ Th "**~{**" and "**~}**" directives delimit an iteration construct that takes one argument, which must be a list. The control string *string* is iterated until all of the elements of the list have been used as arguments to the control string. The number of directives requiring arguments in the control string will be the number of elements fetched from the list argument in a single iteration. If the control string requires **no** arguments, **format** will reiterate *ad infinitum*. Prior to each iteration, the number of remaining elements is checked; iteration is terminated if the list is *nil*. Both the delimiting directives can take modifiers. When the opening directive is "**~{**", the argument must be a list of lists. At each iteration, one sublist is used as the set of arguments consumed by the control string. A new sublist is consumed by each iteration. When the opening directive is "**~@{**", all the remaining arguments to **format** are used as the list of arguments for the control string. When the opening directive is "**~:@{**", all the remaining arguments to **format** are used as the list of arguments for the control string, just as with "**~@{**", but each argument remaining must be a list. Each iteration consumes one list, i.e. one argument to **format**. If the closing directive is "**~:}**", the control string is processed at least once, even if the initial list of arguments available is *nil*, but not if the argument to the iteration directive is 0.

---

```
=> (setq listpos '((car 1) (cadr 2) (caddr 3) (caddr 4)))
((car 1) (cadr 2) (caddr 3) (caddr 4))
=> (format t "~:{function: ~10a position: ~10a~%-}" listpos)
1 1.
function: car          position: 1
```



```

function: cadr      position: 2
function: caddr     position: 3
function: caddr     position: 4
nil
=>
=> (format t "Ships spotted at: ~@:{ <~d , ~d>~} "
          '(2 2) '(4 8) '(3 5) '(7 7))
Ships spotted at: <2 , 2> <4 , 8> <3 , 5> <7 , 7> nil
=>

```

---

□ If the iteration directive does not contain a control string, the next argument to **format** is taken to be the control string. Arguments following the string argument are processed by the iteration. Evaluation of the control string occurs prior to the first iteration.

```

"~mincolw,colincr,minpad,padchar:@<"
"~sepwidth,linewidth;"
"~>"
"~<text1~;;text2~;text3~>"

```

■ The "**~<text>**" directive expression causes *text* to be justified within a field which is at least *mincolw* wide. If the column width given would be insufficient to print *text*, *mincolw* is adjusted upward by *colincr*. The prefix *minpad* is the minimum number of pad characters *padchar* (default is the space character) to output between fields of *text*. Fields of *text*, *text<sub>i</sub>*, are separated by "**~;**" separator directives. The number of columns in the output will be the number of fields in *text*. If no prefixes appear in the "**~<**" directive, the leftmost field is left-justified; the rightmost field is right justified; if there is only one field it is right justified. The **:** modifier causes spacing to be added before the first field is printed; the **@** modifier causes spaces to be added to the last field.

□ If the first field of the "**~<text>**" directive is terminated with a "**~;;**" directive, the remaining fields are first evaluated and the padding for the output (exclusive of the first field) determined. Then, if the resulting padded text will fit on the current line, it is output and the first field is discarded. Otherwise the first field is output first, followed by the padded text. If the first field ends with a "**~;;**" that has a prefix, the padded text must fit on the current line with *sepwidth* characters to spare. A second prefix *linewidth* can be used to specify a line width to use to calculate whether the padded text will fit. For example, to output a list so that each line after the first is indented five spaces and no element runs over the current column boundaries we can use the following string.

```

"~%;; {~<~%;; ~5;; ~s~>~,}.~%"

```



"~first,second,third:~"

■ In all of the directives described so far, an error will be caused if **format** runs out of arguments. Within "~{" directive expressions, the "~" directive causes the expression to be terminated if there are no more arguments to be processed. Within "<" expressions, this directive causes remaining unprocessed fields to be ignored. The "~" directive should only appear at the beginning of a field since it aborts the entire field. When there is no enclosing "<" or "~{" directive, the entire **format** function is aborted (without error).

□ If a prefix is present, then the iteration is stopped if the prefix is zero (hence "~" is equivalent to "~#"). If two prefixes are present, the iteration is stopped if they are **equal**, for example, "~1,#" will halt iteration if only one argument remains. If three prefixes are present, termination occurs if the second is between the first and the third in order of ascendancy. Both # and v are acceptable prefixes.

□ When "~" is used within a "~:" directive, only the **current** iteration is halted; remaining iterations begin with the next argument. To escape from the entire construct, use "~:~".

## 6 System functions

6.1	System functions	6-1
6.2	The Lisp Profiler	6-13





## 6 System functions

### 6.1 System functions

This chapter describes the functions used to interact with internal components of the LISP system and operating system.

#### (allocate 's\_type 'x\_pages)

- WHERE *s\_type* is one of the FRANZ LISP data types described in §1.1 except for port or *hunk*.
- RETURNS *x\_pages*.
- SIDE EFFECT: FRANZ LISP attempts to allocate *x\_pages* of type *s\_type*. If there are fewer than *x\_pages* of memory available for allocation, no space is allocated and an error occurs. The storage that is allocated is not given to the caller. Rather, it is added to the free storage list of *s\_type*. By contrast, the functions **segment** and **small-segment** allocate blocks of storage and return it to the caller.
- NOTE: You cannot allocate additional ports, and although you can allocate more *hunk* space, you must specify which size, e.g. you can allocate more *hunk0*, *hunk1*, . . . , *hunk6* space. (*Hunkn* holds up to  $2^{n+1}$  items).

#### (argv 'x\_argnumb)

- RETURNS a symbol whose pname is the *x\_argnumbth* argument (starting at 0) on the command line that invoked the current LISP.
- NOTE: If *x\_argnumb* is less than zero, a *fixnum* whose value is the number of arguments on the command line is returned. (argv 0) returns the name of the LISP you are running.

#### (baktrace)

- RETURNS *nil*.
- SIDE EFFECT: The LISP runtime stack is examined and the name of (most) of the functions currently in execution are printed, most active first.
- NOTE: This occasionally misses the names of compiled LISP functions due to incomplete information on the stack. If you are tracing compiled code, then **baktrace** is not able to interpret the stack unless (sstatus translink nil) was done. See the function **showstack** for another way of printing the LISP runtime stack. This



misspelling is for compatibility with MacLISP.

(chdir 's\_path)

- RETURNS *t* iff the system call succeeds.
- SIDE EFFECT: The current directory is set to *s\_path*. Among other things, this affects the default location where the input/output functions look for and create files.
- NOTE: **chdir** follows the standard operating system conventions. If *s\_path* does not begin with a slash, the default path is changed to the current path with *s\_path* appended.

(command-line-args)

- RETURNS a list of the arguments typed on the command line either to the LISP interpreter, or saved LISP dump, or application compiled with the autorun option ('liszt -r').

(deref 'x\_addr)

- RETURNS the contents of *x\_addr*, when thought of as a longword memory location.
- NOTE: This may be useful in constructing arguments to 'C' functions out of 'dangerous' areas of memory.

(dumplisp s\_name)

- RETURNS *nil*.
- SIDE EFFECT: The current LISP is dumped to the named file. When *s\_name* is executed, you are in a LISP in the same state as when the **dumplisp** was done.
- NOTE: **dumplisp** fails if you try to write over the current running file. Most operating systems do not allow you to modify the file you are running.

(eval-when l\_time g\_exp1 ...)

- SIDE EFFECT: *l\_time* may contain any combination of the symbols *load*, *eval*, and *compile*. The effects of *load* and *compile* are discussed in §12.3.2.1 on the compiler. If *eval* is present, however, this simply means that the expressions *g\_exp1*, and so on, are evaluated from left to right. If *eval* is not present, the forms are not evaluated.

(exec s\_arg1 ...)

- RETURNS the result of forking and executing the command named by concatenating the *s\_argi* together with spaces in between. Note that this does not just do a simple C **exec()**. The value returned is the exit status of the subprocess.



(exece 's\_fname ['l\_args ['l\_envir]])

■ RETURNS the error code from the system if it was unable to execute the command `s_fname` with arguments `l_args` and with the environment set up as specified in `l_envir`. If `l_envir` is not present, the LISP's process environment is used. If this function is successful, it is not returned, instead the LISP system is overlaid by the new command. The `exece()` Unix system call is used for this function.

■ NOTE: The first argument in `l_args` must be the name of the program to exec (`s_fname`). If `l_args` is not present, `s_fname` is used as the first argument.

(exit ['x\_code])

■ RETURNS nothing (it never returns).

■ SIDE EFFECT: The LISP system dies with exit code `x_code` or 0 if `x_code` is not specified.

(fake 'x\_addr)

■ RETURNS the LISP object at address `x_addr`.

■ NOTE: This is intended to be used by people debugging the LISP system.

(fork)

■ RETURNS *nil* to the child process and the process number of the child to the parent.

■ SIDE EFFECT: A copy of the current LISP system is made in memory, and both LISP systems now begin to run. This function can be used interactively to temporarily save the state of LISP (as shown later), but you must be careful that only one of the LISP's interacts with the terminal after the fork. The `wait` function is useful for this.

---

```
=> (setq foo 'bar) ; Set a variable.
bar
=> (cond ((fork)(wait))) ; Duplicate the Lisp system and
nil                ; make the parent wait.
=> foo ; Check the value of the variable.
bar
=> (setq foo 'baz) ; Give it a new value.
baz
=> foo ; Make sure it worked.
baz
=> (exit) ; Exit the child.
(5274 . 0) ; The wait function returns this.
=> foo ; Check to make sure parent was
bar ; not modified.
```

---



(gc)

- RETURNS *nil*.
- SIDE EFFECT: This causes a garbage collection.
- If the symbol *\$gcprint* is bound to *t*, then a terse summary of space used will be printed.
- NOTE: Pages of type *string* are, by default, not garbage collected since the amount of space retrieved is usually not worth the amount of time spent collecting the pages. If, however, you have an application which generates and releases lots of *strings*, you can include them during the regular garbage collection by executing the function (*sstatus gcstrings t*). To turn off string garbage collection, do (*sstatus gcstrings nil*).
- NOTE: See *gcafter* and *gcbefore* below.

(gcafter *s\_type*)

(gcbefore *s\_type*)

- WHERE *s\_type* is one of the FRANZ LISP data types listed in §1.1.
- NOTE: The function *gcbefore* is called before the execution of the garbage collector when space of type *s\_type* is exhausted. The flag *gcdisable* will be bound to *t* for the duration of the call. The argument to *gcafter* which is called by the garbage collector after the garbage collection is the same data type *s\_type*. Usually the function *gcafter* should determine if more space need be allocated, and, if so, should allocate it. There is a default *gcafter* function, but if you want control over space allocation, you can define your own. However, be sure that it is an *nlambda*.

(getenv '*st\_name*)

- RETURNS the value of looking up *st\_name* in current environment. Not all operating systems have an environment.

(include *s\_filename*)

- RETURNS *nil*.
- SIDE EFFECT: The given filename is loaded into the LISP system.
- NOTE: This is similar to *load* except that the argument is not evaluated. *Include* means something special to the compiler.

(include-if '*g\_predicate* *s\_filename*)

- RETURNS *nil*.
- SIDE EFFECT: This has the same effect as *include* but is only actuated if the predicate is non-*nil*.

(includef '*s\_filename*)

- RETURNS *nil*.



- **SIDE EFFECT:** This is the same as **include** except that the argument is evaluated.

(**include-if** 'g\_predicate s\_filename)

- **RETURNS** *nil*.
- **SIDE EFFECT:** This has the same effect as **include** but is only actuated if the predicate is non-*nil*.

(**maknum** 'g\_arg)

- **RETURNS** the address of its argument converted into a *fixnum*.

(**opval** 's\_arg ['g\_newval])

- **RETURNS** the value associated with s\_arg before the call.
- **SIDE EFFECT:** If g\_newval is specified, the value associated with s\_arg is changed to g\_newval.
- **NOTE:** **opval** keeps track of storage allocation. If s\_arg is one of the data types, then **opval** returns a list of three *fixnums* representing the number of items of that type in use, the number of pages allocated, and the number of items of that type per page. You should never try to change the value that **opval** associates with a data type using **opval**.

(**\*process** 'st\_command ['g\_readp ['g\_writep]])

- **RETURNS** either a *fixnum* if one argument is given, or a list of two ports and a *fixnum* if two or three arguments are given.
- **NOTE:** **\*process** starts another process by passing st\_command to the shell. If

---

On the Tektronix 4404 there are two shells: */bin/shell* and */bin/script*. If the user assigns a value to the operating system's *LispSHELL* environment variable, then that value will be used for the shell.

---

only one argument is given to **\*process**, **\*process** waits for the new process to die and then returns the exit code of the new process. If more than two or three arguments are given, **\*process** starts the process and then returns a list which, depending on the value of g\_readp and g\_writep, may contain I/O ports for communicating with the new process. If g\_writep is non-*nil*, then a port is created that the LISP program can use to send characters to the new process. If g\_readp is non-*nil*, then a port is created that the LISP program can use to read characters from the new process. The value returned by **\*process** is (readport writeport pid), where readport and writeport are either *nil* or a port based on the value of g\_readp and g\_writep. pid is the process id of the new process. Since it is hard to remember the order of g\_readp and g\_writep, the functions **\*process-send** and **\*process-receive** are written to perform the common functions.



**(\*process-receive 'st\_command)**

- RETURNS a port that can be read.
- SIDE EFFECT: The command `st_command` is given to the shell, and it is started running in the background. The output of that command is available for reading via the port returned. The input of the command process is set to `/dev/null`.

**(\*process-send 'st\_command)**

- RETURNS a port that can be written to.
- SIDE EFFECT: The command `st_command` is given to the shell, and it is started running in the background. The LISP program can provide input for that command by sending characters to the port returned by this function. The output of the command process is set to `/dev/null`.

**(process s\_pgrm [s\_frompipe s\_topipe])**

- RETURNS if the optional arguments are not present, a *fixnum* that is the exit code when `s_pgrm` dies. If the optional arguments are present, it returns a *fixnum* that is the process identification of the child.
- NOTE: This command is obsolete. New programs should use one of the **\*process** commands given earlier.
- SIDE EFFECT: If `s_frompipe` and `s_topipe` are given, they are bound to ports that are pipes that direct characters from FRANZ LISP to the new process and to FRANZ LISP from the new process respectively. **process** forks a process named `s_pgrm` and waits for it to die if and only if there are no pipe arguments given.

**(ptime)**

- RETURNS a list of two elements. The first is the amount of processor time used by the LISP system so far, and the second is the amount of time used by the garbage collector so far.
- NOTE: The time is measured in those units used by the `times()` system call, usually 60ths of a second (100ths of a second on the Tektronix 4404). The first number includes the second number. The amount of time used by garbage collection is not recorded until the first call to **ptime**. This is done to prevent overhead when the user is not interested in garbage collection times.

**(reset)**

- SIDE EFFECT: The LISP runtime stack is cleared and the system restarts at the top level.

**(\*rset 'g\_flag)**

- RETURNS `g_flag`.



■ **SIDE EFFECT:** If `g_flag` is non-*nil*, then the LISP system maintains extra information about calls to **eval** and **funcall**. This record keeping slows down the evaluation, but this is required for the functions **evalhook**, **funcallhook**, and **evalframe** to work. To debug compiled LISP code, the transfer tables should be unlinked: (`ssstatus translink nil`).

**(segment 's\_type 'x\_size)**

- WHERE `s_type` is one of the data types given in §1.1.
- RETURNS a segment of contiguous lispvals of type `s_type`.
- NOTE: In reality, **segment** returns a new data cell of type `s_type` and allocates space for `x_size - 1` more `s_type`'s beyond the one returned. **segment** always allocates new space and does so in 512 byte chunks. If you ask for 2 *fixnums*, **segment** actually allocates 128 of them, thus, wasting 126 *fixnums*. The function **small-segment** is a smarter space allocator and should be used whenever possible.

**(shell)**

- RETURNS the exit code of the shell when it dies.
- **SIDE EFFECT:** This forks a new shell and returns when the shell dies.

**(showstack)**

- RETURNS *nil*.
- **SIDE EFFECT:** All forms currently in evaluation are printed, beginning with the most recent. For compiled code, **showstack** reveals only the function name, and it may miss some functions which you might expect from interpreted code.

**(signal 'x\_signum 's\_name)**

- RETURNS *nil* if no handler currently exists for signal `x_signum`; otherwise it returns the handler that was catching the signal.
- **SIDE EFFECT:** This identifies the function named `s_name` to handle the signal number `x_signum`. If `s_name` is *nil*, the signal is ignored.

**(sizeof 'g\_arg)**

- RETURNS the number of bytes required to store one object of type `g_arg`, encoded as a *fixnum*.

**(small-segment 's\_type 'x\_cells)**

- WHERE `s_type` is one of *fixnum*, *flonum*, and *value*.
- RETURNS a segment of `x_cells` data objects of type `s_type`.
- **SIDE EFFECT:** This may call **segment** to allocate new space, or it may be able to fill the request on a page already allocated. The value returned by **small-segment** is usually stored in the data subpart of an array object.



**(sstatus g\_type g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: If *g\_type* is not one of the special **sstatus** codes described in the next few pages, this simply sets *g\_val* as the value of status type *g\_type* in the system status property list.

**(sstatus appendmap g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: If *g\_val* is non-*nil*, when **fasl** is told to create a load map, it appends to the file name given in the **fasl** command rather than creating a new map file. The initial value is *nil*.

**(sstatus automatic-reset g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: If *g\_val* is non-*nil* when an error occurs that no one wants to handle, a **reset** is done instead of entering a primitive internal break loop. The initial value is *t*.

**(sstatus chainatom g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: If *g\_val* is non-*nil* and a **car** or **cdr** of a symbol is done, then *nil* is returned instead of an error being signaled. This only affects the interpreter not the compiler. The initial value is *nil*.

**(sstatus dumpcore g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: If *g\_val* is *nil*, FRANZ LISP tells the operating system that a segmentation violation or bus error should cause a core dump. If *g\_val* is non-*nil* then FRANZ LISP catches those errors and prints a message advising the user to reset.
- NOTE: The initial value for this flag is *nil*, and only those knowledgeable of the inner characteristics of the LISP system should ever set this flag non-*nil*.

**(sstatus evalhook g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: When *g\_val* is non-*nil*, this enables the **evalhook** and **funcallhook** traps in the evaluator. See §15.4 for more details.

**(sstatus feature g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: *g\_val* is added to the (status features) list.



**(sstatus nofeature g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: *g\_val* is removed from the status features list if it is present.

**(sstatus translink g\_val)**

- RETURNS *g\_val*
- SIDE EFFECT: If *g\_val* is *nil*, then all transfer tables are cleared and further calls through the transfer table do not cause the fast links to be set up. If *g\_val* is the symbol *on*, then all possible transfer table entries are linked and the flag is set to cause fast links to be set up dynamically. Otherwise, all that is done is to set the flag to cause fast links to be set up dynamically. The initial value is *nil*.
- NOTE: For a discussion of transfer tables, see §12.8.

**(sstatus uctolc g\_val)**

- RETURNS *g\_val*.
- SIDE EFFECT: If *g\_val* is not *nil*, then all unescaped capital letters in symbols read by the reader is converted to lower case.
- NOTE: This allows FRANZ LISP to be compatible with single case LISP systems (e.g. MacLISP, InterLISP and UCILISP).

**(status g\_code)**

- RETURNS the value associated with the status code *g\_code* if *g\_code* is not one of the special cases given later.

**(status ctime)**

- RETURNS a symbol whose print name is the current time and date.
- EXAMPLE: (status ctime) = |Sun Jun 29 16:51:26 1980|.
- NOTE: This has been made obsolete by **time-string**, described later.

**(status feature g\_val)**

- RETURNS *t* iff *g\_val* is in the status features list.

**(status features)**

- RETURNS the value of the features code, which is a list of features that are present in this system. You add to this list with (sstatus feature *g\_val*) and test if feature *g\_feat* is present with (status feature *g\_feat*) or with **feature-present** described below.



**(feature-present 'g\_exp)**

■ RETURNS *t* or *nil* depending upon the evaluation of the *g\_exp* in the context of the features present in the LISP.

■ NOTE: Feature-present is used as follows:

```
(feature-present 'f) = (memq 'f (status features))  
(feature-present '(not f1))  
(feature-present '(and f1 f2))  
(feature-present '(or f1 f2))
```

□ *g\_exp* evaluation is calculated as follows. If the feature *fi* is on the (status features) list, replace it with a non-*nil* value otherwise with *nil*. Then perform the specified operations ('and,' 'not,' or 'or'), returning the value.

---

```
=> (status features)  
(long-filenames mvr hasht |68k| string Franz franz)  
=> (feature-present 'mvr)  
(mvr hasht |68k| string Franz franz)  
=> (feature-present '(not mvr))  
nil  
=> (feature-present '(or vax 68k)).  
t  
=> (feature-present '(and mvr hasht))  
t
```

---

**(status isatty)**

■ RETURNS *t* iff the standard input is a terminal.

**(status localtime)**

■ RETURNS a list of *fixnums* representing the current time.

■ EXAMPLE: (status localtime) = (3 51 13 31 6 81 5 211 1).

□ The above result means 3rd second, 51st minute, 13th hour (1 p.m), 31st day, month 6 (0 = January), year 81 (0 = 1900), day of the week 5 (0 = Sunday), 211th day of the year with daylight savings time in effect.

**(status syntax s\_char)**

■ NOTE: This function should not be used. See the description of *getsyntax*, in Chapter 7, for a replacement.



**(status undeffunc)**

- RETURNS a list of all functions that transfer table entries point to, but that are not defined at this point.
- NOTE: Some of the undefined functions listed could be arrays which are not yet created.

**(status version)**

- RETURNS a string that is the current LISP version name.
- EXAMPLE: (status version) returns "*Franz Lisp, Opus 43*".

**(sys:chmod 'st\_filename 'xl\_mode)**

- WHERE *st\_filename* is the name of the file whose mode will be set and *xl\_mode* is the mode to set.
- RETURNS *t* if successful. If unsuccessful (e.g. if the user doesn't have the required permissions or if the file doesn't exist), then an error is signalled.
- NOTE: *xl\_mode* can be a *fixnum* representing the mode or it can be a list of symbolic mode descriptors. Use of a *fixnum* mode is discouraged as it is operating system dependent. Only the low 7 bits of the *fixnum* are important. The bits have this meaning:

1	user read permission
2	user write permission
4	user execute permission
8	other read permission
16	other write permission
32	other execute permission
64	other execute permission
128	set user id upon execution

- In order to set a combination of permissions you add the individual permission numbers. For example to get user read, user write and other read permission:  $1+2+8 = 11$ .
- An alternative to the *fixnum* mode is a list of symbolic modes, each symbolic mode describing one or more permission bits.
- Each symbolic mode is a string or symbol whose name is *s* or has the form *[uoa]=[rwx]*. The brackets are meta-syntax denoting the fact that one or more of the enclosing characters can be present. The brackets themselves are not typed. The meaning of the symbolic mode *s* is that the set user id bit should be set. In *[uoa]=[rwx]*, the *u* and *o* stand for user and other, and *a* stands for all, meaning both *u* and *o* (on operating system which support group permission, the *g* option is provided and *a* also includes group permissions). *r* stands for read permission, *w* for write permission and *x* for execute permission. To specify a symbolic bit, you must specify both a class (user, group, other and/or all) and a set of bits (read, write and/or execute). The symbolic mode is a list of these symbolic mode descriptors.



□ If the global symbol *tilde-expansion* is non-*nil*, then the filename will be expanded to include the path-name to the conventional directory.

■ EXAMPLE: To set the mode of the file *foo* to be read/write by the user who owns the file, but only readable by others, use the following: (sys:chmod 'foo '(u=rw o=r)) or (sys:chmod 'foo '(u=r u=w o=r)) or (sys:chmod 'foo '(a=r u=w)).

(sys:access 'st\_filename 'x\_mode)

■ RETURNS true if the file has the given mode.

(sys:getpid)

(sys:link 'st\_oldfilename 'st\_newfilename)

(sys:time)

(sys:unlink 'st\_filename)

■ NOTE: This set of functions is intended to provide symbolic access to various operating system features in a reasonably portable way. Tilde-expansion of filenames is performed on all filename arguments if the global variable *tilde-expansion* is non-*nil*.

(sys:getuid)

■ RETURNS the user identification number of the user running LISP.

■ NOTE: This number uniquely identifies the user on a machine.

(sys:getpwuid 'x\_uid)

■ RETURNS a vector containing the information from the password file on the user whose id is *x\_uid*. It returns *nil* if no user with that id exists.

□ The vector has the following form:

index 0:	user name (a string)
index 1:	user id (a fixnum)
index 2:	group id (a fixnum; <i>nil</i> if this notion is unimplemented)
index 3:	user's login directory (a string)

(sys:getpwnam 'st\_name)

■ RETURNS the same information as **sys:getpwuid** but it searches for a given entry in the password file using the user's name, not the user's id.

(time-string ['x\_seconds])

■ RETURNS an ASCII string giving the time and date that was *x\_seconds* after operating system's idea of creation (Midnight, Jan 1, 1970 GMT). If no argument is given, time-string returns the current date. This should be used rather than (status



ctime), and may be used to make the results of **filestat** more intelligible.

**(top-level)**

- RETURNS nothing (it never returns)
- NOTE: This function is the top-level **read-eval-print** loop. It never returns any value. Its main use is that if you redefine it and do a **reset**, then the redefined **top-level** is invoked.

**(wait)**

- RETURNS a dotted pair (processid . status) when the next child process dies.
- NOTE: This function returns (processid . status) when the child dies or stops (via **ptrace()**). It returns (-1 . status) when no children are alive to wait for (Unix errno is **ECHILD**). It also returns (-1 . status) when an error occurred (**EFAULT**) or a signal is received (**EINTR**). The low-order byte of status is the termination code (either zero or a signal number); the high-order byte is the exit status returned from the process. The low-order byte is -1 on a **ptrace()** stop.

## 6.2 The Lisp Profiler

FRANZ LISP has a facility whereby one can profile code being executed. This profiler uses the *evalhook/funcallhook* mechanism to get control everytime a function is called. It maintains a running count for each function of the functions which call it and the number of times they do the calling.

**(prof-start)**

- SIDE EFFECT: This functions sets up LISP in a state so that every function call gets recorded for the profiler.
- NOTE: one must be careful not to do a (reset) since that will turn off the profiling.

**(prof-end)**

- SIDE EFFECT: This function turns off profiling.

**(prof-report)**

- SIDE EFFECT: This function lists each function that was called, who called this function and who this function calls.



40

Received of the Treasurer of the

of the sum of \$100.00 for the purpose of

the sum of \$100.00 for the purpose of

Wm. H. H. Co.

for the sum of \$100.00 for the purpose of

for the sum of \$100.00 for the purpose of

10/10/18

10/10/18

10/10/18

## 7 The LISP reader

- 7.1 Introduction 7-1
- 7.2 Syntax classes 7-1
- 7.3 Reader operations 7-2
- 7.4 Character classes 7-3
- 7.5 Syntax classes 7-6
- 7.6 Character macros 7-10
  - 7.6.1 Types 7-10
    - 7.6.1.1 Normal 7-10
    - 7.6.1.2 Splicing 7-11
    - 7.6.1.3 Infix 7-11
  - 7.6.2 Invocations 7-12
- 7.7 Functions 7-12





## 7 The LISP reader

### 7.1 Introduction

The **read** function is responsible for converting a stream of characters into a LISP expression. **read** is table driven and the table it uses is called a *readtable*. The **print** function does the inverse of **read**; it converts a LISP expression into a stream of characters. Typically, the conversion is done in such a way that if a stream of characters is read by **read**, the result is an expression equal to the one **print** is given. **print** must also refer to the readtable in order to determine how to format its output. The **explode** function, which returns a list of characters rather than printing them, must also refer to the readtable.

A readtable is created with the **makereadtable** function, modified with the **setsyntax** function and interrogated with the **getsyntax** function. The structure of a readtable is hidden from the user—a readtable should only be manipulated with the three functions mentioned earlier.

There is one distinguished readtable called the *current readtable* whose value determines what **read**, **print**, and **explode** do. The current readtable is the value of the symbol *readtable*. Thus, it is possible to rapidly change the current syntax by *lambda*-binding a different readtable to the symbol *readtable*. When the binding is undone, the syntax reverts to its old form.

### 7.2 Syntax classes

The readtable describes how each of the 128 ASCII characters should be treated by the reader and printer. Each character belongs to a *syntax class*, which has three properties:

*Character class.* Tells what the reader should do when it sees this character. There are a large number of character classes. They are described later.

*Separator.* Most types of tokens the reader constructs are one character long. Four token types have an arbitrary length: number (1234), symbol print name (*franz*), escaped symbol print name (*/franz/*), and string ("*franz*"). The reader can easily determine when it has come to the end of one of the last two types: it just looks for the matching delimiter (*/* or *"*). When the reader is reading a number or symbol print name, it stops reading when it comes to a character with the *separator* property. The separator character is pushed back into the input stream and is the first character read when the reader is called again.



*Escape.* Tells the printer when to put escapes in front of, or around, a symbol whose print name contains this character. There are three possibilities: (1) always escape a symbol with this character in it, (2) only escape a symbol if this is the only character in the symbol, and (3) only escape a symbol if this is the first character in the symbol. (Note that the printer always escapes a symbol which, if printed out, looks like a valid number.)

When the LISP system is built, LISP code is added to a C-coded kernel and the result becomes the standard LISP system. The readtable present in the C-coded kernel, called the *raw readtable*, contains the bare necessities for reading in LISP code. During the construction of the complete LISP system, a copy is made of the raw readtable and then the copy is modified by adding *macro* characters. The result is what is called the *standard readtable*. When a new readtable is created with *makereadtable*, a copy is made of either the raw readtable or the current readtable, which is likely to be the standard readtable.

### 7.3 Reader operations

The reader has a very simple algorithm. It is either *scanning* for a token, *collecting* a token, or *processing* a token. Scanning involves reading characters and throwing away those that do not start tokens, such as blanks and tabs. Collecting means gathering the characters that make up a token into a buffer. Processing may involve creating symbols, strings, lists, *fixnums*, *bignums*, or *flonums*; or calling a user written function called a character *macro*.

The components of the syntax class determine when the reader switches between the scanning, collecting, and processing states. The reader continues scanning as long as the character class of the characters it reads is *cseparator*. When it reads a character whose character class is not *cseparator*, it stores that character in its buffer and begins the collecting phase.

If the character class of that first character is *ccharacter*, *cnumber*, *cperiod*, or *csign*, then it continues collecting until it runs into a character whose syntax class has the *separator* property. (That last character is pushed back into the input buffer and is the first character read next time.) Now, the reader goes into the processing phase, checking to see if the token it reads is a number or symbol. It is important to note that after the first character is collected the component of the syntax class that tells the reader to stop collecting is the *separator* property, not the character class.

If the character class of the character that stopped the scanning is not *ccharacter*, *cnumber*, *cperiod*, or *csign*, then the reader processes that character immediately. The character classes *csingle-macro*, *csingle-splicing-macro*, and *csingle-infix-macro* act like *ccharacter* if the following token is not a *separator*. The processing that is done for a given character class is described in detail in the next section.



## 7.4 Character classes

### *ccharacter (normal character)*

raw readtable: A-Z a-z ^H ! # \$ % & \* , / ; < = > ? @ ^ \_ ' { } ~

standard readtable: A-Z a-z ^H ! \$ % & \* / ; < = > ? @ ^ \_ { } ~

- A normal character.

### *cnumber (digit)*

raw readtable: 0-9

standard readtable: 0-9

- This type is a digit. The syntax for an integer (*fixnum* or *bignum*) is a string of *cnumber* characters optionally followed by a *cperiod*. If the digits are not followed by a *cperiod*, then they are interpreted in base *ibase*, which must be eight or ten. The syntax for a floating point number is either zero or more *cnumbers* followed by a *cperiod* and then followed by one or more *cnumber*. A floating point number may also be an integer or floating point number followed by *e* or *d*, an optional *+* or *-*, and then zero or more *cnumbers*.

### *csign (leading sign)*

raw readtable: + -

standard readtable: + -

- A leading sign for a number. No other characters should be given this class.

### *cpackage (package marker)*

raw readtable: :

standard readtable: :

- The package marker character separates a package name from the name of the external symbol available in that package.

### *cleft-paren (left parenthesis)*

raw readtable: (

standard readtable: (

- A left parenthesis. Tells the reader to begin forming a list.

### *cright-paren (right parenthesis)*

raw readtable: )

standard readtable: )

- A right parenthesis. Tells the reader that it has reached the end of a list.



*cleft-bracket (left bracket)*

raw readtable: [  
 standard readtable: [

- A left bracket. Tells the reader that it should begin forming a list. See the description of *cright-bracket* for the difference between *cleft-bracket* and *cleft-paren*.

*cright-bracket (right bracket)*

raw readtable: ]  
 standard readtable: ]

- A right bracket. A *cright-bracket* finishes the formation of the current list and all enclosing lists until it finds one that begins with a *cleft-bracket* or until it reaches the top level list.

*cperiod (dotted-pair separator)*

raw readtable: .  
 standard readtable: .

- The period is used to separate element of a cons cell; that is, (a . (b . nil)) is the same as (a b). *cperiod* is also used in numbers as described earlier.

*cseparator (token separator)*

raw readtable: ^I^M ESCAPE SPACE  
 standard readtable: ^I^M ESCAPE SPACE

- Separates tokens. When the reader is scanning, these character are passed over. Note: there is a difference between the *cseparator* character class and the *separator* property of a syntax class.

*csingle-quote (quote)*

raw readtable: '  
 standard readtable: '

- This causes **read** to be called recursively and the list (quote <value read>) to be returned.

*csymbol-delimiter (multi-character escape)*

raw readtable: |  
 standard readtable: |

- This causes the reader to begin collecting characters and to stop only when another identical *csymbol-delimiter* is seen. The only way to escape a *csymbol-delimiter* within a symbol name is with a *cescape* character. The collected characters are converted into a string which becomes the print name of a symbol. If a



symbol with an identical print name already exists, then the allocation is not done, rather the existing symbol is used.

*cescape (single-character escape)*

raw readtable: \  
standard readtable: \

■ This causes the next character that is read in to be treated as a *vcharacter*. A character whose syntax class is *vcharacter* has a character class *ccharacter* and does not have the *separator* property so it does not separate symbols.

*cstring-delimiter (string quote)*

raw readtable: "  
standard readtable: "

■ This is the same as *csymbol-delimiter* except that the result is returned as a string instead of a symbol.

*csingle-character-symbol (single-character symbol)*

raw readtable: none  
standard readtable: none

■ This returns a symbol whose print name is the the single character that has been collected.

*cmacro (character macro)*

raw readtable: none  
standard readtable: ' ,

■ The reader calls the *macro* function associated with this character and the current readtable, passing it no arguments. The result of the *macro* is added to the structure the reader is building, just as if that form were directly read by the reader. More details on *macros* are provided later.

*csplicing-macro (splicing character macro)*

raw readtable: none  
standard readtable: #;

■ A *csplicing-macro* differs from a *cmacro* in the way the result is incorporated in the structure the reader is building. A *csplicing-macro* must return a list of forms (possibly empty). The reader acts as if it read each element of the list itself without the surrounding parenthesis.



*csingle-macro (conditional character macro)*

raw readtable: none  
standard readtable: none

- This causes the reader to check the next character. If it is a *cseparator*, then this acts like a *cmacro*. Otherwise, it acts like a *ccharacter*.

*csplicing-macro (conditional splicing character macro)*

raw readtable: none  
standard readtable: none

- This is triggered like a *csingle-macro*. However, the result is spliced in like a *csplicing-macro*.

*cinfix-macro (infix character)*

raw readtable: none  
standard readtable: none

- This differs from a *cmacro* in that the *macro* function is passed a form representing what the reader has read so far. The result of the *macro* replaces what the reader had read so far.

*csingle-infix-macro (single infix character)*

raw readtable: none  
standard readtable: none

- This differs from the *cinfix-macro* in that the *macro* is only triggered if the character following the *csingle-infix-macro* character is a *cseparator*.

*cillegal (illegal character)*

raw readtable: ^@-^G^N^Z^-^\_DELETE  
standard readtable: ^@-^G^N^Z^-^\_DELETE

- The characters cause the reader to signal an error if read.

## 7.5 Syntax classes

The readtable maps each character into a syntax class. The syntax class contains three pieces of information: the character class, whether this is a separator, and the escape properties. The first two properties are used by the reader, the last by the printer (and **explode**). The initial LISP system has the following syntax classes defined. You may add syntax classes with **add-syntax-class**. For each syntax class, the properties of the class and which characters have this syntax class by default are listed. More information about each syntax class can be found under the description of the syntax class's character class.

*vcharacter*

character class: *ccharacter*

raw readtable: A-Z a-z ^H!#\$%&\*,./;<=>?@^\_ '{}~

standard readtable: A-Z a-z ^H!#\$%&\*,./;<=>?@^\_ '{}~

*vnumber*

character class: *cnumber*

raw readtable: 0-9

standard readtable: 0-9

*vsign*

character class: *csign*

raw readtable: + -

standard readtable: + -

*vpackage*

character class: *cpackage*

raw readtable: :

standard readtable: :

*vleft-paren*

character class: *cleft-paren*

raw readtable: (

standard readtable: (

properties: *escape-always, separator*

*vright-paren*

character class: *cright-paren*

raw readtable: )

standard readtable: )

properties: *escape-always, separator*

*vleft-bracket*

character class: *cleft-bracket*

raw readtable: [

standard readtable: [

properties: *escape-always, separator*



*vright-bracket*

character class: *cright-bracket*  
raw readtable: ]  
standard readtable: ]  
properties: *escape-always, separator*

*vperiod*

character class: *cperiod*  
raw readtable: .  
standard readtable: .  
property: *escape-when-unique*

*vseparator*

character class: *cseparator*  
raw readtable: ^I^M ESCAPE SPACE  
standard readtable: ^I^M ESCAPE SPACE  
properties: *escape-always, separator*

*vsingle-quote*

character class: *csingle-quote*  
raw readtable: `  
standard readtable: `  
properties: *escape-always, separator*

*vsymbol-delimiter*

character class: *csingle-delimiter*  
raw readtable: |  
standard readtable: |  
property: *escape-always*

*vescape*

character class: *cescape*  
raw readtable: \  
standard readtable: \  
property: *escape-always*

*vstring-delimiter*

character class: *cstring-delimiter*  
raw readtable: "  
standard readtable: "  
property: *escape-always*

*vsingle-character-symbol*

character class: *csingle-character-symbol*  
raw readtable: none  
standard readtable: none  
property: *separator*

*vmacro*

character class: *cmacro*  
raw readtable: none  
standard readtable: ' ,  
properties: *escape-always, separator*

*vsplicing-macro*

character class: *csplicing-macro*  
raw readtable: none  
standard readtable: # ;  
properties: *escape-always, separator*

*vsingle-macro*

character class: *csingle-macro*  
raw readtable: none  
standard readtable: none  
property: *escape-when-unique*

*vsingle-splicing-macro*

character class: *csingle-splicing-macro*  
raw readtable: none  
standard readtable: none  
property: *escape-when-unique*



*vinfix-macro*

character class: *cinfix-macro*  
raw readtable: none  
standard readtable: none  
properties: *escape-always, separator*

*vsingle-infix-macro*

character class: *csingle-infix-macro*  
raw readtable: none  
standard readtable: none  
property: *escape-when-unique*

*villegal*

character class: *cillegal*  
raw readtable: *^@-^G^N-^Z^-^\_DELETE*  
standard readtable: *^@-^G^N-^Z^-^\_DELETE*  
properties: *escape-always, separator*

## 7.6 Character macros

Character *macros* are user-written functions that are executed during the reading process. The value returned by a character *macro* may or may not be used by the reader, depending on the type of *macro* and the value returned. Character *macros* are always attached to a single character with the **setsyntax** function.

### 7.6.1 Types

There are three types of character *macros*: normal, splicing, and infix. These types differ in the arguments they are given or in what is done with the result they return.

#### 7.6.1.1 Normal

A normal *macro* is passed no arguments. The value returned by a normal *macro* is simply used by the reader as if it had read the value itself. Here is an example of a *macro* that returns the abbreviation for a given state.

---

```
=> (defun stateabbrev nil
      (cdr (assq (read) '((california . ca) (pennsylvania . pa)))))
stateabbrev
=> (setsyntax '\! 'vmacro 'stateabbrev)
t
=> '( ! california ! wyoming ! pennsylvania)
(ca nil pa)
```



---

Notice what happened to *lwyoming*. Since it was not in the table, the associated function returned *nil*. The creator of the *macro* may have wanted to leave the list alone, in such a case, but could not with this type of reader *macro*. The splicing *macro*, described next, allows a character *macro* function to return a value that is ignored.

### 7.6.1.2 Splicing

The value returned from a splicing *macro* must be a list or *nil*. If the value is *nil*, then the value is ignored; otherwise, the reader acts as if it read each object in the list. Usually, the list only contains one element. If the reader is reading at the top level (that is, not collecting elements of list), then it is illegal for a splicing *macro* to return more than one element in the list. The major advantage of a splicing *macro* over a normal *macro* is the ability of the splicing *macro* to return nothing. The comment character (usually *;*) is a splicing *macro* bound to a function which reads to the end of the line and always returns *nil*. Here is the previous example written as a splicing *macro*

---

```
=> (defun stateabbrev nil
      ((lambda (value)
         (cond (value (list value))
               (t nil)))
       (cdr (assq (read)
                  '((california . ca) (pennsylvania . pa))))))
=> (setsyntax '! 'vsplicing-macro 'stateabbrev)
=> '(!pennsylvania ! foo !california)
(pa ca)
=> '!foo !bar !pennsylvania
pa
```

---

### 7.6.1.3 Infix

Infix *macros* are passed a *tconc* structure representing what has been read so far. Briefly, a *tconc* structure is a single list cell whose *car* points to a list and whose *cdr* points to the last list cell in that list. The interpretation by the reader of the value returned by an infix *macro* depends on whether the *macro* is called while the reader is constructing a list or whether it is called at the top level of the reader. If the *macro* is called while a list is being constructed, then the value returned should be a *tconc* structure. The *car* of that structure replaces the list of elements that the reader has been collecting. If the *macro* is called at top level, then it is passed the value *nil*, and the value it returns should either be *nil* or a *tconc* structure. If the *macro* returns *nil*, then the value is ignored and the reader continues to read. If the *macro* returns a *tconc* structure of one element, that is, whose *car* is a list of one element, then that single element is returned as



the value of `read`. If the *macro* returns a `tconc` structure of more than one element, then that list of elements is returned as the value of `read`.

---

```
=> (defun plusop (x)
      (cond ((null x) (tconc nil '\+))
            (t (lconc nil (list 'plus (caar x) (read))))))

plusop
=> (setsyntax '\+ 'v infix-macro 'plusop)
t
=> '(a + b)
(plus a b)
=> '+
|+|
```

---

### 7.6.2 Invocations

There are three different circumstances in which you would like a *macro* function to be triggered.

**Always.** Whenever the *macro* character is seen, the *macro* should be invoked. This is accomplished by using the character classes *cmacro*, *csplicing-macro*, or *cinfix-macro* and by using the *separator* property. The syntax classes *vmacro*, *vsplicing-macro*, and *vsingle-macro* are defined this way.

**When first.** The *macro* should only be triggered when the *macro* character is the first character found after the scanning process. A syntax class for a *when first* macro is defined using *cmacro*, *csplicing-macro*, or *cinfix-macro* but not including the *separator* property.

**When unique.** The *macro* should only be triggered when the *macro* character is the only character collected in the token collection phase of the reader; that is, the *macro* character is preceded by zero or more *cseparators* and followed by a *separator*. A syntax class for a *when unique* macro is defined using *csingle-macro*, *csingle-splicing-macro*, or *csingle-infix-macro* but not including the *separator* property. The syntax classes so defined are *vsingle-macro*, *vsingle-splicing-macro*, and *vsingle-infix-macro*.

### 7.7 Functions

(setsyntax 's\_symbol 's\_synclass ['ls\_func])

■ WHERE `ls_func` is the name of a function or a *lambda* body. If `s_synclass` is *macro* or *splicing*, then `ls_func` is the associated function.

■ RETURNS `t`.

■ SIDE EFFECT: `s_symbol` should be a symbol whose print name is only one character. The syntax class for that character is set to `s_synclass` in the current readtable. If `s_synclass` is a class that requires a character *macro*, then `ls_func` must be



supplied.

**(getsyntax 's\_symbol)**

- RETURNS the syntax class of the first character of `s_symbol`'s print name. `s_symbol`'s print name must be exactly one character long.
- NOTE: This function supersedes (status syntax) that no longer exists.

**(add-syntax-class 's\_synclass 'l\_properties)**

- RETURNS `s_synclass`.
- SIDE EFFECT: Defines the syntax class `s_synclass` to have properties `l_properties`. The list `l_properties` should contain a character class mentioned earlier. `l_properties` may contain one of the escape properties: *escape-always*, *escape-when-unique*, or *escape-when-first*. `l_properties` may contain the *separator* property. After a syntax class has been defined with **add-syntax-class**, the **setsyntax** function can be used to give characters that syntax class.

---

```
; Define a non-separating macro character.  
; This type of macro character is used in UCILisp, and  
; it corresponds to a FIRST MACRO in InterLisp.
```

```
=> (add-syntax-class 'vuci-macro '(cmacro escape-when-first))  
vuci-macro
```

---



1

The first part of the report deals with the general situation in the country.

The second part deals with the specific situation in the various regions.

The third part deals with the specific situation in the various regions.

The fourth part deals with the specific situation in the various regions.

The fifth part deals with the specific situation in the various regions.

The sixth part deals with the specific situation in the various regions.

The seventh part deals with the specific situation in the various regions.

The eighth part deals with the specific situation in the various regions.

The ninth part deals with the specific situation in the various regions.

The tenth part deals with the specific situation in the various regions.

The eleventh part deals with the specific situation in the various regions.

The twelfth part deals with the specific situation in the various regions.

The thirteenth part deals with the specific situation in the various regions.

The fourteenth part deals with the specific situation in the various regions.

The fifteenth part deals with the specific situation in the various regions.

The sixteenth part deals with the specific situation in the various regions.

The seventeenth part deals with the specific situation in the various regions.

The eighteenth part deals with the specific situation in the various regions.

The nineteenth part deals with the specific situation in the various regions.

The twentieth part deals with the specific situation in the various regions.

The twenty-first part deals with the specific situation in the various regions.

The twenty-second part deals with the specific situation in the various regions.

The twenty-third part deals with the specific situation in the various regions.

The twenty-fourth part deals with the specific situation in the various regions.

The twenty-fifth part deals with the specific situation in the various regions.

The twenty-sixth part deals with the specific situation in the various regions.

The twenty-seventh part deals with the specific situation in the various regions.

The twenty-eighth part deals with the specific situation in the various regions.

## 8 Program forms

- 8.1 Valid function objects 8-1
- 8.2 Functions 8-1
- 8.3 Macros 8-6
  - 8.3.1 Macro forms 8-6
  - 8.3.2 Defmacro 8-7
  - 8.3.3 Protection against reëvaluation 8-7
  - 8.3.4 The backquote character macro 8-8
  - 8.3.5 Sharp sign character macros 8-9
    - 8.3.5.1 Conditional inclusion 8-11
    - 8.3.5.2 Read time evaluation 8-12
- 8.4 Closures and fclosures 8-12
  - 8.4.1 An example 8-13
  - 8.4.2 Useful functions 8-14
  - 8.4.3 Fclosures 8-14
- 8.5 Functional objects table 8-15





## 8 Program forms

### 8.1 Valid function objects

There are many different program forms in FRANZ LISP. These objects can occupy the function field of a symbol object and be 'called' by other functions. The traditional forms in earlier versions of FRANZ LISP have been augmented by the modified *lambda* expressions required by COMMON LISP and by *closures*. Various constructions on top of these (such as *Flavors*, described in Chapter 19) are also useful. Table 8.1 at the end of this chapter shows all of the possibilities, how to recognize them, and where to look for documentation.

### 8.2 Functions

The basic technique for defining a function in LISP is to use **def** or **defun** and use normal *lambda* binding. For example

```
(def foo (lambda (x y) ...))
```

or equivalently

```
(defun foo (x y) ...).
```

When a *lambda* function is called, the actual arguments are evaluated from left to right and are *lambda*-bound to the formal parameters of the *lambda* function.

FRANZ LISP provides an *nlambda* function type which is often used for functions that are invoked at top level, and give the program more control over evaluation of arguments. Some built-in functions which evaluate their arguments in special ways are defined in FRANZ LISP as *nlambdas* (for example **cond**, **do**, and **or**). When an *nlambda* function is called, the list of unevaluated arguments is *lambda*-bound to the single formal parameter of the *nlambda* function. For example:

---

```
=> (def foo (nlambda (l) l))  
=> (foo a b c)  
(a b c)
```

---



FRANZ LISP also supplies *lexprs*, which allow for an arbitrary number of evaluated arguments. When a *lexpr* function is called, the arguments are evaluated and a *fixnum*, whose value is the number of arguments, is *lambda*-bound to the single formal parameter of the *lexpr* function. The *lexpr* can then access the arguments using the **arg** function. For example:

---

```
=> (def foo
      (lexpr (nargs)
            (list nargs (arg 1) (arg 2) (arg 3))))
foo
=> (foo 'a 'b 'c)
(3 a b c)
```

---

When a function is compiled, *special* declarations may be needed to preserve the behavior of functions using its *lambda*-bound arguments as “free” or “global” variables. It is important in the case of such access to a dynamic variable to declare the corresponding formal parameter *special* (see §12.3.2.2).

*Lambda* and *lexpr* functions both compile into a binary objects which have the same formal calling discipline, *lambda*.

In order to provide compatibility with Common LISP functions, it is possible to use the expanded *lambda*-expression in a function definition as indicated in the template below. (Literal parts of the template are shown in **bold gothic face** and ***bold italic gothic face*** (the latter for signifiers), metasyntax is shown in *italic gothic face*, and variables are shown in gothic face.)

```
(defun function-name
  ({req-var}*)
  [&optional {opt-var / (opt-var [initform [opt-svar]])}]*)
  [&rest rest-var]
  [&key {key-var / ({key-var / (keyword key-var)} [initform [key-svar]])}]*)
  [&allow-other-keys]
  [&aux {aux-var / (aux-var [initform])}]*)
  {form}*)
```

This admittedly daunting specification follows Common LISP, and the description given in Guy L. Steele’s *Common LISP: The Language* (Digital Press, 1984), referred to as *CLTL* in this chapter. On a first reading, you may prefer to skip to the examples at the end of this section before embarking on the detailed explanation which follows immediately, based on *CLTL*.

Each element of the formal parameter specification list, or the *lambda*-list (that is, the list following the function name above) is either a *parameter specifier* or a *lambda*-list *signifier*. *Lambda*-list signifiers begin with the ampersand character **&**.

In all cases a var or svar must be a symbol, the name of a variable; each keyword must be a keyword symbol which begins with a colon, such as **:start**. An initform may



be any form.

A *lambda*-list has five parts, any or all of which may be empty:

- Specifiers for the **required** parameters: These are all the names req-vars up to the first *lambda*-list signifier; if there is no *lambda*-list signifier, then all the specifiers are for required parameters.
- Specifiers for **&optional** parameters: If the *lambda*-list signifier **&optional** is present, the optional names are those following the *lambda*-list signifier **&optional** up to the next *lambda*-list signifier or the end of the list.
- A specifier for a **&rest** parameter: The *lambda*-list signifier **&rest**, if present, must be followed by a single name rest-var, which in turn may be followed by another *lambda*-list signifier or the end of the *lambda*-list.
- Specifiers for **&key** parameters: If the *lambda*-list signifier **&key** is present, all names key-vars up to the next *lambda*-list signifier or the end of the list are **&key** names. The keyword names may optionally be followed by the *lambda*-list signifier **&allow-other-keys**.
- Specifiers for **&aux** variables: These are not really parameters. If the *lambda*-list signifier **&aux** is present, all names aux-vars after it are *auxiliary variables*.

When the function represented by the *lambda*-expression is applied to arguments, the arguments and parameters are processed in order from left to right. In the simplest and most usual case, only required parameters are present in the *lambda*-list; each is specified simply by a name req-var for the parameter variable. When the function is applied, there must be exactly as many arguments as there are parameters, and each parameter is bound to one argument.

In the more general case, if there are  $n$  required parameters ( $n$  may be zero), there must be at least  $n$  arguments, and the required parameters are bound to the first  $n$  arguments. The other parameters are then processed using any remaining arguments.

If **&optional** parameters are specified, then each one is processed as follows. If any unprocessed arguments remain, then the parameter variable opt-var is bound to the next remaining argument, just as for a required parameter. If no arguments remain, however, then the initform part of the parameter specifier is evaluated, and opt-var is bound to the resulting value (or to *nil* if no initform appears in the parameter specifier). If another variable name opt-svar appears in the specifier, it is bound to *t* if an argument was available, and to *nil* if no argument remained (and therefore initform had to be evaluated). The variable opt-svar is called a supplied-p parameter; it is bound not to an argument but to a value indicating whether or not an argument had been supplied for its associated parameter.

After all **&optional** parameter specifiers have been processed, then there may or may not be a **&rest** parameter. If there is a **&rest** parameter, it is bound to a list of all as-yet-unprocessed arguments. (If no unprocessed arguments remain, the **&rest** parameter is bound to *nil*.) If there is no **&rest** parameter and there are no **&key** parameters, then there should be no unprocessed arguments (it is an error if there are).

Next, any **&key** parameters are processed. For this purpose the same arguments are processed that would be made into a list for a **&rest** parameter. (Indeed, it is permitted to specify both **&rest** and **&key**. In this case the remaining arguments are used for both



purposes; that is, all remaining arguments are made into a list for the **&rest** parameter, and are also processed for the **&key** parameters. This is the only situation in which an argument is used in the processing of more than one parameter specifier.) If **&key** is specified, there must remain an even number of arguments; these are considered as pairs, the first argument in each pair being interpreted as a keyword name and the second as the corresponding value. It is an error for the first object of each pair to be anything but a keyword.

In each keyword parameter specifier must be a name key-var for the parameter variable. If an explicit keyword is specified, then that is the keyword name for the parameter. Otherwise the name key-var serves to indicate the keyword name, in that a keyword with the same name (in the keyword package) is used as the keyword. Thus

```
(defun foo (&key radix (type 'integer)) ...)
```

means exactly the same as

```
(defun foo (&key ((:radix radix)) ((:type type) 'integer)) ...).
```

The keyword parameter specifiers are, like all parameter specifiers, effectively processed from left to right. For each keyword parameter specifier, if there is an argument pair whose keyword name matches that specifier's keyword name (that is, the names are **eq**), then the parameter variable for that specifier is bound to the second item (the value) of that argument pair. If more than one such argument pair matches, it is not an error; the leftmost argument pair is used. If no such argument pair exists, then the initform for that specifier is evaluated and the parameter variable is bound to that value (or to *nil* if no initform was specified). The variable key-svar is treated as for ordinary **&optional** parameters: it is bound to *t* if there was a matching argument pair, and to *nil* otherwise.

It is an error if an argument pair has a keyword name not matched by any parameter specifier, unless at least one of the following two conditions is met:

- **&allow-other-keys** was specified in the *lambda*-list.
- Among the keyword argument pairs is a pair whose keyword is **:allow-other-keys** and whose value is not *nil*.

If either condition occurs, then it is not an error for an argument pair to match no parameter specified, and the argument pair is simply ignored (but such an argument pair is accessible through the **&rest** parameter if one was specified). The purpose of these mechanisms is to allow sharing of argument lists among several functions and to allow either the caller or the called function to specify that such sharing may be taking place.

After all parameter specifiers have been processed, the auxiliary variable specifiers (those following the *lambda*-list keyword **&aux**) are processed from left to right. For each one, the initform is evaluated and the variable aux-var bound to that value (or to *nil* if no initform was specified). Nothing can be done with **&aux**

variables that cannot be done with the special form **let\***, and the following two forms are equivalent: (defun foo (x y &aux (a 1) (b 2) c) ...) and (defun foo (x y) (let\* ((a 1) (b 2) c) ...)).

Whenever any initform is evaluated for any parameter specifier, that form may refer to any parameter variable to the left of the specifier in which the initform appears,



including any supplied-*p* variables, and may rely on the fact that no other parameter variable has yet been bound (including its own parameter variable).

Once the *lambda*-list has been processed, the forms in the body of the *lambda*-expression are executed. These forms may refer to the arguments to the function by using the names of the parameters. On exit from the function, either by a normal return of the function's value(s) or by a non-local exit, the parameter bindings are no longer in effect. (The bindings are not necessarily permanently discarded, for a binding can later be reinstated if a **closure** over that binding was created and saved before the exit occurred).

Examples of **&optional** and **&rest** parameters:

---

```
(defun baz (x &optional (y 2)) (+ x (* y 3)))
(baz 4 5) = 19
(baz 4) = 10
(defun foo (&optional (x 2 y) (z 3 w) &rest q) (list x y z w q))
(foo) = (2 nil 3 nil nil)
(foo 6) = (6 t 3 nil nil)
(foo 6 3) = (6 t 3 t nil)
(foo 6 3 8) = (6 t 3 t (8))
(foo 6 3 8 9 10 11) = (6 t 3 t (8 9 10 11))
```

---

Examples of **&key** parameters:

---

```
(defun fum (u &key c d) (list u c d))
(fum 1) = (1 nil nil)
(fum 1 :c 6) = (1 6 nil)
(fum 1 :d 8) = (1 nil 8)
(fum 1 :c 6 :d 8) = (1 6 8)
(fum 1 :d 8 :c 6) = (1 6 8)
(fum :u :d 8 :c 6) = (:u 6 8)
(fum :u :c :d) = (:u :d nil)
```

---

Examples of mixtures:

---

```
(defun fie (a &optional (b 3) &rest x &key c (d a))
  (list a b c d x))
(fie 1) = (1 3 nil 1 nil)
(fie 1 2) = (1 2 nil 1 nil)
(fie :c 7) = (:c 7 nil :c nil)
(fie 1 6 :c 7) = (1 6 7 1 (:c 7))
(fie 1 6 :d 8) = (1 6 nil 8 (:d 8))
(fie 1 6 :d 8 :c 9 :d 10) = (1 6 9 8 (:d 8 :c 9 :d 10))
```

---



All symbols whose names begin with **&** are conventionally reserved for use as *lambda*-list signifiers and should not be used as variable names. (Note: In *CLTL*, the phrase *lambda-list keyword* is confusingly used for signifier.)

### 8.3 Macros

An important feature of LISP is its ability to manipulate programs as data. As a result of this, most LISP implementations have very powerful *macro*-definition facilities. The FRANZ LISP language's *macro* facility can be used to incorporate popular features of other languages into LISP. For example, there are *macro* packages that allow you to create records (as in Pascal) and refer to elements of those records by the field names. The **defstruct** package imported from MacLISP does this. Another popular use for *macros* is to create more readable control structures which expand into **cond**, **or**, and **and**.

#### 8.3.1 Macro forms

A *macro* is a function that accepts a LISP expression as input and returns another LISP expression. The action the *macro* takes is called *macro* expansion. Here is a simple example:

---

```
=> (def first (macro (x) (cons 'car (cdr x))))  
first  
=> (first '(a b c))  
a  
=> (apply 'first '(first '(a b c)))  
(car '(a b c))
```

---

The first input line defines a *macro* called **first**. Notice that the *macro* has one formal parameter, **x**. On the second input line, you ask the interpreter to evaluate **(first '(a b c))**. **eval** sees that **first** has a function definition of type *macro*, so it evaluates **first**'s definition, passing to **first**, as an argument, the form **eval** itself was trying to evaluate: **(first '(a b c))**. The **first macro** discards the **car** of the argument with **cdr**, **conses** a **car** at the beginning of the list and returns **(car '(a b c))**, which **eval** evaluates. The value **a** is returned as the value of **(first '(a b c))**. Thus, whenever **eval** tries to evaluate a list whose **car** has a *macro* definition, it ends up doing (at least) two operations: the first of which is a call to the *macro* to let it *macro* expand the form, and the second of which is the evaluation of the result of the *macro*. The result of the *macro* may be yet another call to a *macro*, so **eval** may have to do even more evaluations until it can finally determine the value of an expression. One way to see how a *macro* expands is to use **apply** as shown on the third input line earlier.



### 8.3.2 Defmacro

The *macro* **defmacro** makes it easier to define *macros* because it allows you to name the arguments to the *macro* call. For example, suppose you find yourself often writing code like (setq stack (cons newelt stack)). You could define a *macro* named **push** to do this. One way to define it is:

```
(def push
  (macro (x) (list 'setq (caddr x) (list 'cons (cadr x) (caddr x)))))
```

then (push newelt stack) expands to the form mentioned earlier. The same *macro* written using **defmacro** would be:

```
(defmacro push (value stack)
  (list 'setq ,stack (list 'cons ,value ,stack)))
```

**Defmacro** allows you to name the arguments of the *macro* call and makes the *macro* definition look more like a function definition.

### 8.3.3 Protection against reevaluation

**defmacro** in FRANZ LISP takes a **&protect** option which allows the programmer to declare that one or more of the arguments should be protected against re-evaluation.

Consider, for example:

```
(defmacro printtwice (x) '(progn (print ,x) (print ,x)))
```

If we execute (printtwice "foo") it would print "foo""foo", probably just what we intended, since it expands to (progn (print "foo")(print "foo")).

Something more mysterious happens if we execute

```
(setq i 0)
(printtwice (setq i (1+ i)))
```

The expression printed is 12 instead of 11. In this case the *printtwice* *macro* expands to (progn (print (setq i (1+ i))) (print (setq i (1+ i)))) which increments *i* twice.

Any argument to a *macro* may be protected against re-evaluation by using the **&protect** keyword as follows:

```
(defmacro printtwice (x &protect (x))
  '(progn (print ,x) (print ,x)))
```

The *macro* expansion will then evaluate *x* once. If the evaluation of *x* could possibly have side effects (i.e. *x* is not an atom), the evaluation will be done once, and the value *lambda*-bound. This makes it simple to use a *macro* to replace a *lambda*. In compiled code there is no time penalty in using **&protect** if the protected arguments are atoms.



### 8.3.4 The backquote character macro

The default syntax for FRANZ LISP has four characters with associated *character macros*. One is semicolon for comments. Two others are the backquote and comma, which are used by the backquote *character macro*. The fourth is the sharp sign *macro* described in the next section.

The backquote *macro* is used to create lists where many of the elements are fixed (quoted). This makes it very useful for creating *macro* definitions. In the simplest case, a backquote acts just like a single quote:

---

```
=> `(a b c d e)
(a b c d e)
```

---

If a comma precedes an element of a backquoted list, then that element is evaluated and its value is put in the list.

---

```
=> (setq d '(x y z))
(x y z)
=> `(a b c ,d e)
(a b c (x y z) e)
```

---

If a comma, followed by an at sign, precedes an element in a backquoted list, then that element is evaluated and spliced into the list with **append**.

---

```
=> `(a b c ,@d e)
(a b c x y z e)
```

---

Once a list begins with a backquote, the commas may appear anywhere in the list as this example shows:

---

```
=> `(a b (c d , (cdr d)) (e f (g h ,@(cddr d) ,@d)))
(a b (c d (y z)) (e f (g h z x y z)))
```

---

One convention for splicing in a list as the last item of a bigger list is to use dotted pair notation as shown below:



---

```
=> `(a b . ,d)
(a b x y z)
```

---

It is also possible, and sometimes even useful, to use the backquote *macro* within itself. As a final demonstration of the backquote *macro*, define the first and push *macros* using all the power at your disposal: **defmacro** and the backquote *macro*.

---

```
=> (defmacro first (list) `(car ,list))
first
=> (defmacro push (value stack)
      `(setq ,stack (cons ,value ,stack)))
stack
```

---

### 8.3.5 Sharp sign character macros

The sharp sign *macro* can perform a number of different functions at read time. The character directly following the sharp sign determines which function is done, and the following LISP *s-expressions* may serve as arguments.

```
(defsharp g_key l_arglist g_exp1 ...)
```

- WHERE *g\_key* is either a single character or a list of single characters.
- SIDE EFFECT: A sharp sign character *macro* for the character(s) in *g\_key* and with argument list *l\_arglist* is defined to call *g\_exp1* ...
- Sharp sign *macros* are invoked by a two character sequence, consisting of the sharp sign (#), followed by an additional character, which will be discussed shortly. Listed below are the *macros*, listed by the two character sequences:

#'

- This is an abbreviation of function. *#'foo* is read as (function foo).

#(

- Reads the following forms, up to a right parenthesis, into a LISP vector. The forms are evaluated.



**#,**  
**#.**

■ The following form is evaluated at read time. See discussion below.

**#\**

■ If the form after the **#\** is a one of *newline, space, rubout, page, tab, backspace, return, linefeed, vert, sharp, lpar, or rpar* then the character code for the above is read. And otherwise, the form must be a single character, and the form read is the character code for that character.

**#|**  
**|#**

■ The characters read between **#|** and **|#** are discarded. This form does nest, so **#|** **|#** **|#** **|#** is valid.

**#+**  
**#-**

■ The second following form is read (or not read) depending on whether (or not) the following form is on the (status features) list. See discussion below.

**#:**

■ The syntax **#:symb** denotes an uninterned symbol whose name is *symb*. The symbol name *symb* must be unqualified (no embedded colons). Each occurrence of this syntax creates a different uninterned symbol.

**#s**  
**#S**

■ The syntax **#s(struct slot1 value1 slot2 value2 . . .)** denotes a structure. This syntax is valid only if *struct* is the name of a structure that is already defined by **defstruct** and which has a standard constructor *macro*. If *construct* is the name of the constructor *macro*, the **#s** syntax above is equivalent to **#. (construct keyword1 'value1 keyword2 'value2 . . .)**, where each keyword<sub>n</sub> is the result of evaluating (intern (string slot<sub>n</sub>) 'keyword).

**#o**  
**#O**

■ Reads the following form as an octal number.



#x  
#X

- Reads the following form as a hexadecimal number.

#/ c

- Returns the *fixnum* representation of the character c.

#^ c

- returns the *fixnum* representation of the control character c, e.g. #^D.

### 8.3.5.1 Conditional inclusion

If you plan to run one source file in more than one environment, then you may want some pieces of code to be included or not included depending on the environment. The 'C' language uses "#ifdef" and "#ifndef" for this purpose, and LISP uses #+ and #- . The environment that the sharp sign *macro* checks is the (status features) list, which is initialized when the LISP system is built and which may be altered by (sstatus feature foo) and (sstatus nofeature bar).

#+ when what

- where when is either a symbol or an expression involving symbols and the functions **and**, **or**, and **not**. The meaning is that what is only read in if when is true. A symbol in when is true only if it appears in the (status features) list.

---

```
; Suppose you want to write a program that references a file
; and that can run at ucb, ucsd, and cmu where the file naming
; conventions are different.
;
=> (defun howold (name)
      (terpr)
      (load #+(or ucb ucsd) "/usr/lib/lisp/ages.l"
            #+cmu "/usr/lisp/doc/ages.l")
      (patom name)
      (patom " is ")
      (print (cdr (assoc name agefile)))
      (patom "years old")
      (terpr))
```

---



#- when what

■ EQUIVALENT TO: #+(not when) what.

### 8.3.5.2 Read time evaluation

Occasionally you want to express a constant as a LISP expression, yet you do not want to pay the penalty of evaluating this expression each time it is referenced. The form

#.expression

evaluates the expression at read time and returns its value.

---

```
; Here is a function to test if any of bits 1, 3 or 12 are set
; in a fixnum.
=> (defun testit (num)
      (cond ((zerop (boole 1 num #.(+ (lsh 1 1) (lsh 1 3) (lsh 1 12))))
            (nil)
            (t t)))
```

---

## 8.4 Closures and fclosures

A *closure* is a type of functional object useful for implementing advanced control structures and data access mechanisms. Its purpose is to remember the values of some variables between invocations of the functional object and to protect this data from being inadvertently overwritten by other LISP functions. This environment can then be re-used. Closures in FRANZ LISP provide the same functionality as *closures* in the MIT LISP Machine LISP (ZetaLISP) design. Closures are also used as the implementation basis for **flavors** (described in chapter 19).

The form (closure var-list function), where var-list is a list of variables and function is any function, creates and returns a *closure*. When this *closure* is applied to some arguments, all of the value cells of the variables on var-list are saved away, and the value cells that those variables had *at the time closure was called* are made to be the value cells of the symbols. Then function is applied to the arguments. (This paragraph is somewhat complex, but it completely describes the operation of *closures*.)

*Fclosures*, a simplified version of *closures* may be available in your version of FRANZ LISP. In later releases, *fclosures* are preserved (although with a different and more efficient underlying implementation strategy) for compatibility.

*Fclosures* are related to *closures* in this way:

(fclosure '(a b) 'foo) ≡ (let ((a a) (b b)) (closure '(a b) 'foo))

We proceed with an example in the next section.



### 8.4.1 An example

---

```
% lisp
Franz Lisp, Opus 43.1
; define a function which creates even number generators
=> (defun new-even-generator (seed)
      (closure '(seed)
                '(lambda ()
                    (cond ((oddp seed) (setq seed (+ 1 seed)))
                          (t (setq seed (+ 2 seed)))))))

new-even-generator
=> (setq e0 (new-even-generator 0))
#<closure #x10ae08>
; e0 is a functional object which will return a new even number each
; time it is called, starting at 0.
=> (setq e11 (new-even-generator 11))
#<closure #x10ae18>
; e11, like e0, will return a new even each time, but
; e11 starts after 11
=> (funcall e0)
2
=> (setq seed 234)
234
; setting seed will have no effect on the operation of e0 or e11.
=> (funcall e0)
4
=> (funcall e11)
12
=> (funcall e11)
14
=> seed
234
```

---

The function **closure** creates a new object called a *closure*, containing a functional object including an environment in which variables (symbols) and the values exist.

When a *closure* is **funcalled**:

- 1 The LISP system *lambda*-binds the symbols in the *closure* to their values in the *closure*.
- 2 It continues the **funcall** on the functional object of the *closure*.
- 3 Finally, it *un-lambda* binds the symbols in the *closure* and at the same time stores the current values of the symbols in the *closure*.



### 8.4.2 Useful functions

(closure 'l\_vars 'g\_funcobj)

- RETURNS a *closure*.
- WHERE *l\_vars* is a list of symbols (not containing *nil*); *g\_funcobj* is any object that can be **funcalled**. (Objects that can be **funcalled** include compiled LISP functions, *lambda* expressions, symbols, foreign functions, etc.)
- NOTE: In general, if you want a compiled function to be closed over a variable, you must declare the variable to be special within the function. Another example is:

```
(closure '(a b) #'(lambda (x) (plus x (setq a (plus a 1)))))
```

Here, the *#'* construction is a special kind of quote which, if run through the compiler indicates that this should be compiled as a function rather than a quoted symbolic expression.

Other functions imported from LISP Machine LISP are the following.

(symeval-in-closure 'cl\_a 's\_x)

- RETURNS the binding of the symbol *s\_x* in the *closure* *cl\_a*. You can look around inside a *closure* with this.

(set-in-closure 'cl\_a 's\_symbol 'g\_x)

- SIDE EFFECT: This sets the binding of *s\_symbol* in the environment of the *closure* *cl\_a* to *g\_x*.

(let-closed argument-list function-body)

- SIDE EFFECT: This is best described by an example:

```
(let-closed ((a 5) b (c 'x)) (function (lambda() ...)))
```

expands into

```
(let ((a 5) b (c 'x))  
  (closure '(a b c) function (lambda() ...))))
```

### 8.4.3 Fclosures

An *fclosure* is a functional object that admits some data manipulations. Historically, FRANZ LISP had *fclosures* before *closures*, and they are preserved for compatability. *Fclosures* are constructed from *vectors*.

*Fclosures* are related to *closures* in this way:

```
(fclosure '(a b) 'foo) = (let ((a a) (b b)) (closure '(a b) 'foo))
```



(fclosure 'l\_vars 'g\_funobj)

- WHERE `l_vars` is a list of variables; `g_funobj` is any object that can be funcalled (including fclosures).
- RETURNS A vector that is the fclosure.

(fclosure-alist 'v\_fclosure)

- RETURNS an association list representing the variables in the fclosure. This is a snapshot of the current state of the fclosure. If the bindings in the fclosure are changed, any previously calculated results of **fclosure-alist** do not change.

(fclosure-function 'v\_fclosure)

- RETURNS the functional object part of the fclosure.

(fclosurep 'v\_fclosure)

- RETURNS `t` if the argument is an fclosure.

(fclosure-list 'l\_vars1 'g\_fcncobj1 [... ...])

- RETURNS a list of fclosures where variables where the same names are shared between fclosures.
- NOTE: **fclosure-list** creates a set of fclosures which share variables. If you create fclosures with the **fclosure** function, they will not share closed-over variables. The following example shows the difference between **fclosure-list** and a pair of **fclosure** calls.

(symeval-in-fclosure 'v\_fclosure 's\_symbol)

- RETURNS the current binding of a particular symbol in an fclosure.

(set-in-fclosure 'v\_fclosure 's\_symbol 'g\_newvalue)

- RETURNS `g_newvalue`.
- SIDE EFFECT: The variable `s_symbol` is bound in the fclosure to `g_newvalue`.

## 8.5 Functional objects table

The following table describes the types of functional objects found in FRANZ LISP and where to locate the documentation for them.



Table 8.1		
informal name	object type	documentation
interpreted lambda function	list with <b>car</b> <b>eq</b> to lambda	8.2
interpreted nlambda function	list with <b>car</b> <b>eq</b> to nlambda	8.2
interpreted lexpr function	list with <b>car</b> <b>eq</b> to lexpr	8.2
interpreted macro	list with <b>car</b> <b>eq</b> to macro	8.3
fclosure	vector with <b>vprop</b> <b>eq</b> to fclosure	8.4
closure	vector with <b>vprop</b> <b>eq</b> to closure	8.4
compiled lambda or lexpr function	binary with discipline <b>eq</b> to lambda	8.2
compiled nlambda function	binary with discipline <b>eq</b> to nlambda	8.2
compiled macro	binary with discipline <b>eq</b> to macro	8.3
foreign subroutine	binary with discipline of "subroutine"	18.
foreign function	binary with discipline of "function"	18
foreign integer function	binary with discipline of "integer-function"	18
foreign real function	binary with discipline of "real-function"	18
foreign 'C' int function	binary with discipline of "c-function"	18
foreign 'C' double function	binary with discipline of "double-c-function"	18
foreign 'C' structure function	binary with discipline of "vector-c-function"	18
foreign 'C' void function	binary with discipline of "void-c-function"	18
array	array object	9
package	package	17

## 9 Arrays and vectors

- 9.1 General arrays 9-1
- 9.2 Subparts of an array object 9-2
  - 9.2.1 Access function 9-2
  - 9.2.2 Auxiliary 9-2
  - 9.2.3 Data 9-2
  - 9.2.4 Length 9-2
  - 9.2.5 Delta 9-3
- 9.3 The MacLISP-compatible array package 9-3
- 9.4 Vectors 9-4
- 9.5 Anatomy of vectors 9-4
  - 9.5.1 Size 9-4
  - 9.5.2 Property 9-5
  - 9.5.3 Internal order 9-5
- 9.6 Immediate vectors 9-5





## 9 Arrays and vectors

Arrays and vectors are two means of expressing aggregate data objects in FRANZ LISP. Vectors may be thought of as sequences of data. They are intended as a vehicle for user-defined data types. This use of vectors is still experimental and subject to revision. As a simple data structure, they are similar to *hunks* and strings. Vectors are used to implement *closures* and are useful to communicate with foreign functions. Both of these topics are discussed in Chapter 8.

Arrays in FRANZ LISP provide a programmable data structure access mechanism. One possible use for FRANZ LISP arrays is to implement MacLISP style arrays, which are simple vectors of *fixnums*, *flonums*, or general LISP values. This is described in more detail in §9.3, but first how array references are handled by the LISP system is described.

The structure of an array object is given in §1.1.11 and reproduced here.

Subpart name	Get value	Set value	Type
access function	<b>getaccess</b>	<b>putaccess</b>	binary, list or symbol
auxiliary	<b>getaux</b>	<b>putaux</b>	lispval
data	<b>arrayref</b>	<b>replace set</b>	block of contiguous lispval
length	<b>getlength</b>	<b>putlength</b>	fixnum
delta	<b>getdelta</b>	<b>putdelta</b>	fixnum

### 9.1 General arrays

Suppose the evaluator is told to evaluate (foo a b) and the function cell of the symbol foo contains an array object, which is called foo\_arr\_obj. First, the evaluator evaluates and stacks the values of a, and b. Next, it stacks the array object foo\_arr\_obj. Finally, it calls the access function of foo\_arr\_obj. The access function should be a *lexpr*<sup>1</sup> or a symbol whose function cell contains a *lexpr*. The access function is responsible for locating and returning a value from the array. The array access function is free to interpret the arguments as it wishes. The MacLISP compatible array access function, which is provided in

<sup>1</sup>A *lexpr* is a function that accepts any number of arguments, which are evaluated before the function is called.



the standard FRANZ LISP system, interprets the arguments as subscripts in the same way as languages like Fortran and Pascal.

The array access function also is called upon to store elements in the array. For example, (store (foo a b) c) automatically expands to (foo c a b), and, when the evaluator is called to evaluate this, it evaluates the arguments c, b, and a. Then it stacks the array object, which is stored in the function cell of foo, and calls the array access function with (now) four arguments. The array access function must be able to tell this is a store operation, which it can do by checking the number of arguments it has been given. (A *lexpr* can do this very easily.)

## 9.2 Subparts of an array object

An array is created by allocating an array object with **marray** and filling in the fields. Certain LISP functions interpret the values of the subparts of the array object in special ways as described in the following text. Placing illegal values in these subparts may cause the LISP system to fail.

### 9.2.1 Access function

The purpose of the access function has been described earlier. The contents of the access function should be a *lexpr*: either a binary (compiled function) or a list (interpreted function). It may also be a symbol whose function cell contains a function definition. This subpart is used by **eval**, **funcall**, and **apply** when evaluating array references.

### 9.2.2 Auxiliary

This can be used for any purpose. If it is a list and the first element of that list is the symbol **unmarked\_array**, then the data subpart is not marked by the garbage collector. Note that this is used in the MacLISP compatible array package and has the potential for causing strange errors if used incorrectly.

### 9.2.3 Data

This is either *nil* or points to a block of data space allocated by segment or small-segment.

### 9.2.4 Length

This is a *fixnum* whose value is the number of elements in the data block. This is used by the garbage collector and by **arrayref** to determine if your index is in bounds.



### 9.2.5 Delta

This is a *fixnum* whose value is the number of bytes in each element of the data block. This is four for an array of *fixnums* or value cells and eight for an array of *flonums*. This is used by the garbage collector and `arrayref` as well.

## 9.3 The MacLISP-compatible array package

A MacLISP style array is similar to what is known as an array structure in other languages: a block of homogeneous data elements that is indexed by one or more integers called subscripts. The data elements can be all *fixnums*, *flonums*, or general LISP objects. An array is created by a call to the function `array` or `*array`. The only difference is that `*array` evaluates its arguments. The call `(array foo t 3 5)` sets up an array called `foo` of dimensions 3 by 5. The subscripts are zero-based. The first element is `(foo 0 0)`, the next is `(foo 0 1)` and so on up to `(foo 2 4)`. The `t` indicates a general LISP object array, which means each element of `foo` can be any type. Each element can be any type since all that is stored in the array is a pointer to a LISP object, not the object itself. `array` does this by allocating an array object with `marray` and then allocating a segment of 15 consecutive value cells with `small-segment` and storing a pointer to that segment in the data subpart of the array object. The length and delta subpart of the array object are filled in (with 15 and 4 respectively) and the access function subpart is set to point to the appropriate array access function. In this case, there is a special access function for two dimensional value cell arrays called `arrac-twoD`, and this access function is used. The auxiliary subpart is set to `(t 3 5)` which describes the type of array and the bounds of the subscripts. Finally, this array object is placed in the function cell of the symbol `foo`. Now when `(foo 1 3)` is evaluated, the array access function is invoked with three arguments: 1, 3, and the array object. From the auxiliary field of the array object it gets a description of the particular array. It then determines which element `(foo 1 3)` refers to and uses `arrayref` to extract that element.

Since this is an array of value cells, what `arrayref` returns is a value cell whose value is what is wanted, so the value cell is evaluated and it is returned as the value of `(foo 1 3)`.

In MacLISP, the call `(array foo fixnum 25)` returns an array whose data object is a block of 25 memory words. When *fixnums* are stored in this array, the actual numbers are stored instead of pointers to the numbers as is done in general LISP object arrays. This is efficient under MacLISP but inefficient in FRANZ LISP since every time a value was referenced from an array it had to be copied and a pointer to the copy returned to prevent aliasing.<sup>2</sup> Thus *t*, *fixnum*, and *flonum* arrays are all implemented in the same manner. This should not affect the compatibility of MacLISP and FRANZ LISP. If there is an application where a block of *fixnums* or *flonums* is required, then exactly the same effect of *fixnum* and *flonum* arrays in MacLISP can be achieved by using *fixnum-block*

<sup>2</sup>Aliasing happens when two variables share the same storage location. For example, if the copying mentioned were not done, then, after `(setq x (foo 2))` was done, the value of `x` and `(foo 2)` would share the same location. Then should the value of `(foo 2)` change, `x`'s value would change as well. This is considered dangerous and, as a result, pointers are never returned into the data space of arrays.



and *flonum-block* arrays. Such arrays are required if you want to pass a large number of arguments to a Fortran- or 'C'-coded function and then get answers back.

The MacLISP compatible array package is just one example of how a general array scheme can be implemented. Another type of array you can implement is the hashed array. The subscript can be anything, not just a number. The access function hashes the subscript and uses the result to select an array element. With the generality of arrays also comes extra cost; if you just want a simple aggregate of less than 128 general LISP objects, you would be wise to look into using *hunks*.

## 9.4 Vectors

Vectors were invented to fix two shortcomings of *hunks*. They can be longer than 128 elements. They also have a tag associated with them, which is intended to say, for example, "Think of me as a *Blobit*." Thus, a vector is an arbitrarily sized *hunk* with a property list.

Continuing the example, the LISP kernel may not know how to print out or evaluate *blobits*, but this is information that is common to all *blobits*. On the other hand, for each individual blobit, there are particulars that are likely to change: height, weight, or eye-color. This is the part that would previously have been stored in the individual entries in the *hunk* and are stored in the data slots of the vector. Here is a summary of the structure of a vector in tabular form:

Subpart name	Get value	Set value	Type
datum[i]	<b>vref</b>	<b>vset</b>	lispval
property	<b>vprop</b> <b>vget</b>	<b>vsetprop</b> <b>vputprop</b>	lispval
size	<b>vsize</b>	—	fixnum

Vectors are created specifying size and optional fill value using the function (new-vector 'x\_size [g\_fill [g\_prop]]) or by initial values: (vector [g\_val ...]).

## 9.5 Anatomy of vectors

There are some technical details about vectors that you should know.

### 9.5.1 Size

You are not free to alter this. It is noted when the vector is created and is used by the garbage collector. The garbage collector coalesces two free vectors, which are neighbors in the heap. The vector size, as reported by the **vsize** functions, return the number of LISP objects in the vector.



### 9.5.2 Property

For other LISP objects which are implemented using vectors, the property list is reserved for internal use. For example, a package object has a special property list, which indicates how it is to be printed. Normally, property lists for vectors follow the same rules as do symbols.

### 9.5.3 Internal order

In memory, vectors start with a longword containing the size, which is immediate data within the vector. The next cell contains a pointer to the property. Any remaining cells, if any, are for data. Vectors are handled differently from any other object in FRANZ LISP in that a pointer to a vector is a pointer to the first data cell, that is, a pointer to the third longword of the structure. This was done for efficiency in compiled code and for uniformity in referencing immediate-vectors (described later). You should never return a pointer to any other part of a vector because this may cause the garbage collector to follow an invalid pointer.

## 9.6 Immediate vectors

Immediate-vectors are similar to vectors. However, they differ in that binary data are stored in space directly within the vector. Thus, the garbage collector preserves the vector itself, if used, and only traverses the property cell. The data may be referenced as single- or double-precision floating-point numbers, longwords, shortwords, or even bytes. Shorts and bytes are returned sign-extended. Since FRANZ LISP does not have single-precision floating-point numbers (*flonums* are double-precision), single-precision floating-point immediate vectors contain *fixnums* in so far as FRANZ LISP is concerned. The compiler open-codes such references, and avoids boxing the resulting data, where possible. Thus, immediate vectors may be used for efficiently processing character data. They are also useful in storing results from functions written in other languages.



Subpart name	Get value	Set value	Type
datum[i]	vrefi-byte vrefi-word vrefi-long vrefi-float vrefi-double	vseti-byte vseti-word vseti-long vseti-float vseti-double	fixnum fixnum fixnum fixnum flonum
property	vprop	vsetprop vputprop	lispval
size	vsize vsize-byte vsize-word vsize-float vsize-double	—	fixnum fixnum fixnum fixnum fixnum

To create immediate vectors specifying size and fill data, you can use the functions **new-vectori-byte**, **new-vectori-word**, **new-vectori-long**, **new-vectori-float**, or **new-vectori-double**. You can also use the functions **vectori-byte**, **vectori-word**, **vectori-long**, **vectori-float**, or **vectori-double**. All of these functions are described in Chapter 2.

## 10 Exception handling

- 10.1 Errset and error handler functions 10-1
- 10.2 The anatomy of an error 10-1
- 10.3 Error handling algorithm 10-2
- 10.4 Default aids 10-3
- 10.5 Autoloading 10-3
- 10.6 Interrupt processing 10-3





## 10 Exception handling

### 10.1 Errset and error handler functions

FRANZ LISP allows you to handle in a number of ways the errors that arise during computation. One way is through the use of the **errset** function. If an error occurs during the evaluation of the **errset**'s first argument, then the locus of control returns to the **errset** which returns *nil* (except in special cases, such as **err**). The other method of error handling is through an error-handler function. When an error occurs, the error handler is called and is given as an argument a description of the error that just occurred. The error handler may take one of the following actions:

- 1 It could take some drastic action like a **reset** or a **throw**.
- 2 It could, if that the error is continuable, return to the function that noticed the error. The error handler indicates that it wants to return a value from the error by returning a list whose **car** is the value it wants to return.
- 3 It could decide not to handle the error and return a non-list to indicate this fact.

### 10.2 The anatomy of an error

Each error is described by a list of these items:

- 1 *Error type*—This is a symbol that indicates the general classification of the error. This classification may determine which function handles this error.
- 2 *Fixnum id*—a *fixnum* identifying the error. In the future each error will have a unique number.
- 3 *Continuable*—If this is non-*nil*, then this error is continuable.
- 4 *Message string*—This is a symbol whose print name is a message describing the error.
- 5 *Data*—There may be from zero to three LISP values that help describe this particular error. For example, the unbound variable error contains one datum value, the symbol whose value is unbound. The list describing that error might look like:

```
(ER%misc 0 t |Unbound Variable:| foobar)
```



### 10.3 Error handling algorithm

This is the sequence of operations when an error occurs:

- 1 If the symbol *ER%all* has a non-*nil* value, then this value is the name of an error-handler function. That function is called with a description of the error. If that function returns (and, of course, it may choose not to) and the value is a list and this error is continuable, then the **car** of the list to the function which called the error is returned. Presumably, the function uses this value to retry the operation. On the other hand, if the error handler returns a non-list, then it has chosen not to handle this error, which leads to step 2. Something special happens before the *ER%all* error-handler is called, which does not happen in any of the other cases described later. To help insure that infinitely recursive errors do not occur, if *ER%all* is set to a bad value, the value of *ER%all* is set to *nil* before the handler is called. Thus, it is the responsibility of the *ER%all* handler to 'reënable' itself by storing its name in *ER%all*.
- 2 Next, the specific error handler for the type of error that just occurred is called, if one exists, to see if it wants to handle the error. The names of the handlers for the specific types of errors are stored as the values of the symbols whose names are the types. For example, the handler for miscellaneous errors is stored as the value of *ER%misc*. Of course, if *ER%misc* has a value of *nil*, then there is no error handler for this type of error. Appendix B contains a list of all error types. The process of classifying the errors is not complete, and, thus, most errors are lumped into the *ER%misc* category. Just as in step 1, the error handler function may choose not to handle the error by returning a non-list, which leads to step 3.
- 3 Next, a check is made to see if there is an **errset** surrounding this error. If so the second argument to the **errset** call is examined. If the second argument was not given or is non-*nil* then the error message associated with this error is printed. Finally, the stack is popped to the context of the **errset** and then the **errset** returns *nil*. If there was no **errset** step 4 is executed.
- 4 If the symbol *ER%tpl* has a value, then it is the name of an error handler that is called in a manner similar to that discussed earlier. If it chooses not to handle the error, step 5 is executed.
- 5 At this point, it has been determined that you do not want to handle this error. Thus, the error message is printed out and a **reset** is done to send the flow of control to the top-level.

To summarize the error handling system: When an error occurs, you have two chances to handle it before the search for an **errset** is done. Then, if there is no **errset**, you have one more chance to handle the error before control jumps to the top level. Every error handler works in the same way: It is given a description of the error (as described in the previous section). It may or may not return. If it returns, then it returns either a list or a non-list. If it returns a list and the error is continuable, then the **car** of the list is returned to the function that noticed the error. Otherwise, the error handler has decided not to handle the error.



## 10.4 Default aids

There are two standard error handlers that probably handle the needs of most users. One of these is the LISP-coded function **tpl-err-tpl-fcn**, which is the default value of *ER%tpl*. Thus, when all other handlers have ignored an error, **tpl-err-tpl-fcn** takes over. It prints out the error message and goes into a read-eval-print loop. The other standard error handler is **tpl-err-all-fcn**. This handler is designed to be connected to *ER%all* and is useful if your program uses **errset** and you want to look at the error before it is thrown up to the **errset**.

## 10.5 Autoloading

When **eval**, **apply**, or **funcall** are told to call an undefined function, an *ER%undef* error is signaled. The default handler for this error is **undef-func-handler**. This function checks the property list of the undefined function for the indicator, *autoload*. If it is present, the value of that indicator should be the name of the file that contains the definition of the undefined function. **Undef-func-handler** loads the file and check if it has defined the function which caused the error. If it has, the error handler returns and the computation continues as if the error did not occur. This provides a way for you to tell the LISP system about the location of commonly used functions. The trace package sets up an autoload property to point to */usr/lib/lisp/trace*.

## 10.6 Interrupt processing

The operating system provides one user-interrupt character that defaults to **^C**.<sup>1</sup> You may select a LISP function to run when an interrupt occurs. Since this interrupt could occur at any time and, in particular, could occur at a time when the internal stack pointers are in an inconsistent state, the processing of the interrupt may be delayed until a safe time. When the first **^C** is typed, the LISP system sets a flag that an interrupt has been requested. This flag is checked at safe places within the interpreter and in the **qlinker** function. If the LISP system does not respond to the first **^C**, another **^C** should be typed. This causes all of the transfer tables to be cleared, forcing all calls from compiled code to go through the **qlinker** function where the interrupt flag is checked. If the LISP system still does not respond, a third **^C** causes an immediate interrupt. This interrupt is not necessarily in a safe place, so the user should **reset** the LISP system as soon as possible.

<sup>1</sup> Actually there are two but the LISP system does not allow you to catch the QUIT interrupt.



The first part of the document is a letter from the President of the United States to the Congress. It is dated January 1, 1861, and is addressed to the House of Representatives. The letter is signed by James Buchanan, the President of the United States at the time. The letter discusses the state of the Union and the recent events leading up to the secession of the Southern states. It is a formal and official document.

The second part of the document is a letter from the Secretary of the United States to the Congress. It is dated January 1, 1861, and is addressed to the House of Representatives. The letter is signed by William B. Ewing, the Secretary of the United States at the time. The letter discusses the state of the Union and the recent events leading up to the secession of the Southern states. It is a formal and official document.

The third part of the document is a letter from the Secretary of the United States to the Congress. It is dated January 1, 1861, and is addressed to the House of Representatives. The letter is signed by William B. Ewing, the Secretary of the United States at the time. The letter discusses the state of the Union and the recent events leading up to the secession of the Southern states. It is a formal and official document.

The fourth part of the document is a letter from the Secretary of the United States to the Congress. It is dated January 1, 1861, and is addressed to the House of Representatives. The letter is signed by William B. Ewing, the Secretary of the United States at the time. The letter discusses the state of the Union and the recent events leading up to the secession of the Southern states. It is a formal and official document.

# 11 The Lister trace package

- 11.1 The trace function 11-1
  - 11.1.1 Trace function argument forms 11-1
  - 11.1.2 Using trace options 11-3
- 11.2 Trace special variables 11-4
- 11.3 Other trace package functions 11-5





## 11 The Lister trace package

The Lister Trace package is an important tool for the interactive debugging of a LISP program. It allows you to examine selected calls to a function or functions, and optionally to stop execution of the LISP program to examine the values of variables.

The trace package is a set of LISP programs located in the LISP program library (usually in the file `/usr/lib/lisp/trace.l`). Although not normally loaded in the LISP system, the package is loaded when the first call to **trace** is made.

### 11.1 The trace function

**(trace [ls\_arg1 ...])**

- WHERE the form of the `ls_argi` is described later.
- RETURNS a list of the function successfully modified for tracing. If no arguments are given to **trace**, a list of all functions currently being traced is returned.
- SIDE EFFECT: The definitions of the functions indicated in the argument list are (usually temporarily) modified.
- NOTE: The form of the arguments is described in the section that follows.

#### 11.1.1 Trace function argument forms

The arguments to the **trace** function can take any of the forms below.

**foo**

- When `foo` is entered and exited, the trace information is printed.

**(foo break)**

- When `foo` is entered and exited, the trace information is printed. Also, just after the trace information for `foo` is printed upon entry, you are put in a special break loop. The prompt is `Tc{1}` and you may type any LISP expression and see its value printed. The *i*th argument to the function just called can be accessed as `(arg i)`. To leave the trace loop, just type `?ret` and execution continues.



**(foo if expression)**

■ When *foo* is entered and the expression evaluates to non-*nil*, then the trace information is printed for both exit and entry. If expression evaluates to *nil*, then no trace information is printed.

**(foo ifnot expression)**

■ When *foo* is entered and the expression evaluates to *nil*, then the trace information is printed for both entry and exit. If both *if* and *ifnot* are specified, then the *if* expression must evaluate to non-*nil* and the *ifnot* expression must evaluate to *nil* for the trace information to be printed out.

**(foo evalin expression)**

■ When *foo* is entered and after the entry trace information is printed, expression is evaluated. Exit trace information is printed when *foo* exits.

**(foo evalout expression)**

■ When *foo* is entered, entry trace information is printed. When *foo* exits, and before the exit trace information is printed, expression is evaluated.

**(foo evalinout expression)**

■ This has the same effect as (trace (foo evalin expression evalout expression)).

**(foo lprint)**

■ This tells **trace** to use the level printer when printing the arguments to and the result of a call to *foo*. The level printer prints only the top levels of list structure. Any structure below three levels is printed as an &. This allows you to trace functions with massive arguments or results.

□ Ordinarily the output from the trace package is printed with *prinlevel* bound to *trace-prinlevel* (default 4) and *prinlength* bound to *trace-prinlength* (default 5). *prinlevel* and *prinlength*, which are useful in cutting off verbose or infinite (cyclical) structures, are special symbols described in Appendix B. If you wish to always print full lists then setting *trace-prinlevel* and *trace-prinlength* each to *nil*, will accomplish this.

□ The following **trace** options permit you to have greater control over each action that takes place when a function is traced. These options are only meant to be used by programmers who need special hooks into the trace package. Most programmers should skip reading this section.



**(foo traceenter tefunc)**

■ This tells **trace** that the function to be called when *foo* is entered is *tefunc*. *tefunc* should be a *lambda* of two arguments. The first argument is bound to the name of the function being traced, *foo* in this case. The second argument is bound to the list of arguments to which *foo* should be applied. The function *tefunc* should print some sort of “entering *foo*” message. It should not apply *foo* to the arguments, however. That is done later on.

**(foo traceexit txfunc)**

■ This tells **trace** that the function to be called when *foo* is exited is *txfunc*. *txfunc* should be a *lambda* of two arguments. The first argument is bound to the name of the function being traced, *foo* in this case. The second argument is bound to the result of the call to *foo*. The function *txfunc* should print some sort of “exiting *foo*” message.

**(foo evfcn evfunc)**

■ This tells **trace** that the form *evfunc* should be evaluated to get the value of *foo* applied to its arguments. This option is a bit different from the other special options since *evfunc* is usually an expression, not just the name of a function, and that expression is specific to the evaluation of function *foo*. The argument list to be applied is available as T-arglist.

**(foo printargs prfunc)**

This tells

■ **trace** to use *prfunc* to print the arguments to be applied to the function *foo*. *prfunc* should be a *lambda* of one argument. You may want to use this option if you want a print function which can handle circular lists. This option works only if you do not specify your own **traceenter** function. Specifying the option *lprint* is just a simple way of changing the *printargs* function to the level printer.

**(foo printres prfunc)**

■ This tells **trace** to use *prfunc* to print the result of evaluating *foo*. *prfunc* should be a *lambda* of one argument. This option works only if you do not specify your own **traceexit** function. Specifying the option *lprint* changes *printres* to the level printer.

**11.1.2 Using trace options**

You may specify more than one option for each function traced. For example:

```
(trace (foo if (eq 3 (arg 1)) break lprint)
      (bar evalin (print xyzzy)))
```

This tells **trace** to trace two more functions, *foo* and *bar*. Should *foo* be called with the first argument **eq** to 3, then the entering *foo* message is printed with the level printer.



Next it enters a trace break loop, allowing you to evaluate any LISP expressions. When you exit the trace break loop, *foo* is applied to its arguments and the resulting value is printed, again using the level printer. *bar* is also traced, and each time *bar* is entered, an "entering *bar*" message is printed and then the value of *xyzy* is printed. Next *bar* is applied to its arguments and the result is printed. If you tell **trace** to trace a function that is already traced, it first **untraces** it. Thus, if you want to specify more than one trace option for a function, you must do it all at once. The following is **not** equivalent to the preceding call to **trace** for *foo*:

```
(trace (foo if (eq 3 (arg 1)))  
      (foo break) (foo lprint))
```

In this example, only the last option, *lprint*, is in effect.

If the symbol *\$tracemute* is given a non-*nil* value, printing of the function name and arguments on entry and exit is suppressed. This is particularly useful if the function you are tracing fails after many calls to it. In this case, you would tell **trace** to trace the function, set *\$tracemute* to *t*, and begin the computation. When an error occurs, you can use **tracedump** to print out the current trace frames.

Generally, the trace package has its own internal names for the LISP functions it uses, so that you can feel free to trace system functions like **cond** and not worry about adverse interaction with the actions of the trace package. You can trace any type of function: *lambda*, *nlambda*, *lexpr*, or *macro*, whether compiled or interpreted, and you can even trace array references. However, you should not attempt to store in an array that has been traced.

When you are tracing compiled code, keep in mind that many function calls are translated directly to machine language or other equivalent function calls. A full list of open-coded functions is listed at the beginning of the Liszt compiler source. **trace** does a (*sstatus translink nil*) to insure that the new traced definitions it defines are called instead of the old untraced ones. You may notice that compiled code runs slower after this is done.

## 11.2 Trace special variables

### *trace-prinlevel*

- the maximum nesting level to print lists. Beyond that point, lists are abbreviated to &.

### *trace-prinlength*

- the maximum length to print a list. Beyond that point, lists are abbreviated to &.

### 11.3 Other trace package functions

**(traceargs s\_func [x\_level])**

- WHERE if `x_level` is missing, it is assumed to be 1.
- RETURNS the arguments to the `x_level`th call to traced function `s_func` are returned.

**(tracedump)**

- SIDE EFFECT: The currently active trace frames are printed on the terminal. It returns a list of functions untraced.

**(untrace [s\_arg1 ...])**

- RETURNS a list of the functions that were untraced.
- NOTE: If no arguments are given, all functions are untraced.
- SIDE EFFECT: The old function definitions of all traced functions are restored except in the case where it appears that the current definition of a function was not created by trace.





## 12 Liszt, the LISP compiler

- 12.1 General strategy of the compiler 12-1
- 12.2 Running the compiler 12-1
- 12.3 Special forms 12-1
  - 12.3.1 Macro expansion 12-1
  - 12.3.2 Classification 12-2
    - 12.3.2.1 Eval-when 12-2
    - 12.3.2.2 Declare 12-2
    - 12.3.2.3 Progn 12-3
    - 12.3.2.4 Include and includef 12-3
    - 12.3.2.5 Def 12-4
    - 12.3.2.6 Other forms 12-4
- 12.4 Using the compiler 12-4
- 12.5 Compiler options 12-5
- 12.6 Autorun 12-7
- 12.7 Pure literals 12-8
- 12.8 Transfer tables 12-8
- 12.9 Fixnum functions 12-9





## 12 Liszt, the LISP compiler

### 12.1 General strategy of the compiler

The purpose of the LISP compiler, Liszt, is to create an object module that, when brought into the LISP system using **fasl**, has the same effect as bringing in the corresponding LISP-coded source module with **load** with one important exception: functions are defined as sequences of machine language instructions instead of LISP *s-expressions*. Liszt is not a function compiler; it is a *file* compiler. Such a file can contain more than function definitions; it can contain other LISP *s-expressions*, which are evaluated at **load** time. These other *s-expressions* are also stored in the object module produced by Liszt and are evaluated at **fasl** time.

As is almost universally true of LISP compilers, the main pass of Liszt is written in LISP.

### 12.2 Running the compiler

The compiler is normally run in this manner:

```
% liszt foo
```

This compiles the file *foo.l* or *foo*. (The preferred way to indicate a LISP source file is to end the file name with *.l*.) The result of the compilation is placed in the file *foo.o*, if no fatal errors were detected. All messages that Liszt generates go to the standard output. Normally each function name is printed before it is compiled. (However, the **-q** option suppresses this.)

### 12.3 Special forms

Liszt makes one pass over the source file. It processes each form in the manner described in the following sections.

#### 12.3.1 Macro expansion

If the form is a *macro* invocation (that is, it is a list whose **car** is a symbol whose function binding is a *macro*), then that *macro* invocation is expanded. This is repeated until the top-level form is no longer a *macro* invocation. When Liszt begins, there are already



some *macros* defined, in fact some functions, such as **defun**, are actually *macros*. You may define your own *macros* as well. For a *macro* to be used, it must be defined in the LISP system in which Liszt runs.

### 12.3.2 Classification

After all *macro* expansion is done, the form is classified according to its **car**. If the form is not a list, then it is classified as an *other* form.

#### 12.3.2.1 Eval-when

The form of **eval-when** is

```
(eval-when (time1 time2 ...) form1 form2 ...)
```

where the *time<sub>i</sub>* are one of *eval*, *compile*, or *load*. The compiler examines the *form<sub>i</sub>* in sequence and the action taken depends on what is in the time list. If *compile* is in the list then the compiler invokes **eval** on each *form<sub>i</sub>* as it examines it. If *load* is in the list, then the compiler recursively calls itself to compile each *form<sub>i</sub>* as it examines it. Note that if *compile* and *load* are in the time list, then the compiler both evaluates and compiles each form. This is useful if you need a function to be defined in the compiler at both compile time, perhaps to aid *macro* expansion, and at run time after the file is **fasled** in.

#### 12.3.2.2 Declare

**declare** is used to provide information about functions and variables to the compiler. It is (almost) equivalent to (eval-when (compile) ...).

You may declare functions to be one of three types: *lambda* (\*expr), *nlambda* (\*fexpr), or *lexpr* (\*lexpr). The names in parenthesis are the MacLISP names and are accepted by the compiler as well, and not just when the compiler is in MacLISP mode. Functions are assumed to be *lambdas* until they are declared otherwise or are defined differently. The compiler treats calls to *lambdas* and *lexprs* equivalently, so you need not worry about declaring *lexprs* either. It is important to declare *nlambdas* before the compiler encounters a call to them. This can be done either via the **declare** directive, or will be done implicitly by the compiler when compiling the definition of an *nlambda*.

Another attribute you can declare for a function is *localf*, which makes the function 'local.' A local function's name is known only to the functions defined within the file itself. The primary advantage of a local function is that its entry and exit protocol is simpler and faster. Short functions can be speed up considerably by declaring them *localfs*. Because the local functions are not known outside the file, these functions can have the same names as functions defined in another file, without a name conflict. This can be a benefit or a hazard. Compiled local functions cannot be used from the functions **funcall** or **apply**.

Variables may be declared *special* or *unspecial*. When a *special* variable is *lambda* bound, either in a **lambda**, **prog**, or **do** expression, its old value is stored away on a stack



for the duration of the **lambda**, **prog**, or **do** expression. This takes time and is often not necessary. Therefore, the default classification for variables is *unspecial*. Space for *unspecial* variables is dynamically allocated on a stack. An *unspecial* variable can only be accessed from within the function where it is created by its presence in a **lambda**, **prog**, or **do** expression variable list. It is possible to declare that all variables are *special* as will be shown later.

You may declare any number of things in each **declare** expression. A sample declaration is:

```
(declare
  (lambda func1 func2)
  (vector-bounds-chk nil)
  (*fexpr func3)
  (*lexpr func4)
  (localf func5)
  (special var1 var2 var3)
  (unspecial var4))
```

You may also declare all variables to be *special* with (declare (specials t)). You may declare that *macro* definitions should be compiled as well as evaluated at compile time by (declare (macros t)). In fact, as was mentioned earlier, declare is much like (eval-when (compile) ...). Thus, if the compiler sees (declare (foo bar)) and foo is defined, then it evaluates (foo bar). If foo is not defined, then an undefined **declare** attribute warning is issued.

If the compiler sees the declaration (declare (vector-bounds-chk nil)), code for checking vector boundaries will not be generated. The default for the compiler is to generate such checking code.

### 12.3.2.3 Progn

When the compiler sees

```
(progn 'compile form1 form2 . . . formn)
```

this it simply compiles form1 through formn as if they too were seen at top-level. One use for this is to allow a *macro* at top-level to expand into more than one function definition for the compiler to compile.

### 12.3.2.4 Include and includef

**include** and **includef** cause another file to be read and compiled by the compiler. The result is the same as if the included file were textually inserted into the original file. The only difference between **include** and **includef** is that **include** does not evaluate its argument and **includef** does. Nested includes are allowed.



### 12.3.2.5 Def

A *def* form is used to define a function. The *macros* **defun** and **defmacro** expand to a *def* form. If the function being defined is a *lambda*, *nlambda*, or *lexpr*, then the compiler converts the LISP definition to a sequence of machine language instructions. If the function being defined is a *macro*, then the compiler evaluates the definition—thus defining the *macro* within the running LISP compiler. Furthermore, if the variable *macros* is set to a non-*nil* value, then the *macro* definition also is translated to machine language and, thus, is defined when the object file is **fasl**ed in. The variable *macros* is set to *t* by **(declare (macros t))**.

When a function or *macro* definition is compiled, *macro* expansion is done whenever possible. If the compiler can determine that a form would be evaluated if this function were interpreted, then it macro-expands it. It does not macro-expand arguments to an *nlambda* unless the characteristics of the *nlambda* are known, as is the case with **cond**. The map functions (**map**, **mapc**, **mapcar**, and so on) are expanded to a **do** statement. This allows the first argument to the **map** function to be a *lambda* expression that references local variables of the function being defined.

### 12.3.2.6 Other forms

All other forms are simply stored in the object file and are evaluated when the file is **fasl**ed in.

## 12.4 Using the compiler

The previous section describes exactly what the compiler does with its input. Generally, you do not have to worry about all that detail because files that work interpreted, work compiled. The following is a list of steps you should follow to insure that a file compiles correctly.

- Make sure all *macro* definitions precede their use in functions or other *macro* definitions. If you want the *macros* to be around when you **fasl** in the object file, you should include this statement at the beginning of the file:

**(declare (macros t)).**

- Make sure all *nlambdas* are defined or declared before they are used. If the compiler comes across a call to a function that has not been defined in the current file, that does not currently have a function binding, and whose type has not been declared, then it assumes that the function needs its arguments evaluated. That is, it is a *lambda* or *lexpr* and generates code accordingly. This means that you do not have to declare *nlambda* functions like **status** since they have an *nlambda* function binding.
- Locate all variables that are used for communicating values between functions. These variables must be declared *special* at the beginning of a file. In most cases, there aren't many *special* declarations, but, if you fail to declare a variable *special* that should be declared, references to those variables which are used 'free' will not access the expected values. Examining the compiler listing will provide indications



of variables used 'free' but not declared *special*. You may eliminate all such messages by adding declarations. Unusual constructions calling interpreted code with 'free' variables can still fail if called from compiled code in which those variables are not declared *special*. Here is an example. Assume that a file contains just these three lines:

```
(def aaa (lambda (glob loc) (bbb loc)))
(def bbb (lambda (myloc) (add glob myloc)))
(def ccc (lambda (glob loc) (bbb loc)))
```

You can see that if you load in these two definitions, then `(aaa 3 4)` is the same as `(add 3 4)` and gives us 7. Suppose you compile the file containing these definitions. When Liszt compiles `aaa`, it assumes that both `glob` and `loc` are local variables and allocates space on the temporary stack for their values when `aaa` is called. Thus, the values of the local variables `glob` and `loc` do not affect the values of the symbols `glob` and `loc` in the LISP system. Now, Liszt moves on to function `bbb`. `myloc` is assumed to be local. When it sees the `add` statement, it finds a reference to a variable called `glob`. This variable is not a local variable to this function, and, therefore, `glob` must refer to the value of the symbol `glob`. Liszt automatically declares `glob` to be *special*, and it prints a warning to that effect. Thus, subsequent uses of `glob` always refer to the symbol `glob`. Next, Liszt compiles `ccc` and treats `glob` as a *special* and `loc` as a local. When the object file is **fasl**ed in and `(ccc 3 4)` is evaluated, the symbol `glob` is lambda-bound to 3, `bbb` is called and returns 7. However, `(aaa 3 4)` fails since when `bbb` is called, `glob` is unbound. What should be done here is to put `(declare (special glob))` at the beginning of the file.

- Make sure that all calls to **arg** are within the *lexpr* whose arguments they reference. If `foo` is a compiled *lexpr* and it calls `bar`, then `bar` cannot use **arg** to get at `foo`'s arguments. If both `foo` and `bar` are interpreted, this works however. The *macro* **listify** can be used to put all or some of a *lexpr*'s arguments in a list, which can then be passed to other functions.

## 12.5 Compiler options

The compiler recognizes a number of options that are described later. The options are typed anywhere on the command line preceded by a minus sign. The entire command line is scanned and all options recorded before any action is taken. Thus

```
% liszt -mx foo
% liszt -m -x foo
% liszt foo -mx
```

are all equivalent. The meanings of the options are:

- A Select an alternate assembler to assemble the file generated by the compiler. If the assembler takes options, one can specify them as

```
% liszt -A "/bin/as -m68010"
```



- C The assembler language output of the compiler is commented. This is useful when debugging the compiler and is not normally done since it slows down compilation.
- I The next command line argument is taken as a filename and loaded prior to compilation.
- e Evaluate the next argument on the command line before starting compilation. For example,

```
% liszt -e '(setq foobar "foo string")' foo
```

evaluates the earlier *s-expression*. Note that the shell requires that the arguments be surrounded by single quotes.

- m Compile this program in MacLISP mode. The reader syntax is changed to the MacLISP syntax and a file of *macro* definitions is loaded in, usually named */usr/lib/lisp/machacks*.
- o Select a different object or assembler language file name. For example,

```
% liszt foo -o xxx.o
```

compiles *foo* into *xxx.o* instead of the default *foo.o*, and

```
% liszt bar -S -o xxx.s
```

compiles to assembly language into *xxx.s* instead of *bar.s*.

- q Run in quiet mode. The names of functions being compiled and various "Note" comments are not printed.
- Q Print compilation statistics and warn of strange constructs. This is the inverse of the -q switch and is the default.
- r Place bootstrap code at the beginning of the object file, which, when the object file is executed, causes a LISP system to be invoked and the object file *fasl*ed in. This is known as 'autorun' and is described later.
- a Annotate the compiled functions. This flag will cause LISP to put information on the compiled functions' symbols' property lists indicating how many arguments the function takes, and the file in which the function is defined.
- S Create an assembly language file only.

```
% liszt -S foo
```

Creates the assembly language file *foo.s* but does not attempt to assemble it. If this option is not specified, the assembly language file is put in the temporary disk area under an automatically generated name based on the LISP compiler's process id. Then, if there are no compilation errors, the assembler is invoked to assemble the file.

- T Print the assembly language output on the standard output file. This is useful when debugging the compiler.
- u Run in UCILISP mode. The character syntax is changed to that of UCILISP and a UCILISP-compatibility package of *macros* is read in.
- w Suppress warning messages.



-x Create a cross reference file.

```
% liszt -x foo
```

not only compiles *foo* into *foo.o* but also generates the file *foo.x*. The file *foo.x* is LISP-readable and lists for each function all functions which that function could call. The program *lxref* reads one or more of these human-readable cross reference listing.

## 12.6 Autorun

The object file that Liszt writes does not contain all the functions necessary to run the LISP program, which was compiled. In order to use the object file, a LISP system must be started and the object file **fasled** in. When the **-r** switch is given to Liszt, the object file created contains a small piece of bootstrap code at the beginning, and the object file is made executable. Now, when the name of the object file is given to the operating system command interpreter (shell) to run, the bootstrap code at the beginning of the object file causes a LISP system to be started. The first action the LISP system takes is to **fasl** in the object file that started it. In effect, the object file has created an environment in which it can run.

Autorun is an alternative to **dumplisp**. The advantage of autorun is that the object file that starts the whole process is typically small, whereas the minimum **dumplisped** file is very large. The disadvantage of autorun is that the file must be **fasled** into a LISP system each time it is used, whereas the file which **dumplisp** creates can be run as is. Liszt itself is a **dumplisped** file since it is used so often and is large enough that too much time is spent **fasling** it in each time it is used. The LISP cross reference program, *lxref*, uses **autorun**, since it is a small and rarely used program.

In order to have the program **fasled** in, begin execution (rather than starting a LISP top-level), the value of the symbol *user-top-level* should be set to the name of the function to get control. An example of this is shown next.

---

```
# Suppose you want to replace the operating system
# date program with one written in LISP.
% cat lisptime.l
(defun mydate nil
  (patom "The date is ")
  (patom (status ctime))
  (terpr)
  (exit 0))
(setq user-top-level 'mydate)
% liszt -r lisptime
Compilation begins with Liszt 68000 version 12.11.1
(C) Copyright 1985, Franz Inc., Alameda Ca.
source: lisptime.l, result: lisptime.o
mydate
%Note: lisptime.l: Compilation complete
%Note: lisptime.l: Time: Real: 0:3, CPU: 0:0.28, GC: 0:0.00 for 0 gcs
```



```
%Note: lisptime.1: Assembly begins
%Note: lisptime.1: Assembly completed successfully
# This changes the name to remove the ".o", (this isn't necessary).
% mv lisptime.o lisptime
# This tests it out.
% lisptime
The date is Sat Aug 1 16:58:33 1984
%
```

---

## 12.7 Pure literals

Normally, the quoted LISP objects (literals) that appear in functions are treated as constants. Consider this function:

```
(def foo
  (lambda nil
    (cond
      ((not (eq 'a (car (setq x '(a b)))))
        (print 'impossible!!))
      (t (rplaca x 'd)))))
```

At first glance it seems that the first **cond** clause is never true, since the **car** of (a b) should always be a. However, if you run this function twice, it prints *"impossible!!"* the second time. This is because the following clause modifies the 'constant' list (a b) with the **rplaca** function. Such modification of literal LISP objects can cause programs to behave strangely as the earlier example shows, but, more importantly, it can cause garbage collection problems if done to compiled code. When a file is **fasl**ed in, if the symbol *\$purcopylits* is non-*nil*, the literal LISP data is put in 'pure' space; that is, it is put in space that need not be looked at by the garbage collector. This reduces the work the garbage collector must do, but it is dangerous, since if the literals are modified to point to non-pure objects, the marker may not mark the non-pure objects. If the symbol *\$purcopylits* is *nil*, then the literal LISP data is put in impure space and the compiled code acts like the interpreted code when literal data is modified. The default value for *\$purcopylits* is *t*.

## 12.8 Transfer tables

A transfer table is setup by **fasl** when the object file is loaded in. There is one entry in the transfer table for each function that is called in that object file. The entry for a call to the function **foo** has two parts whose contents are:

- **Function address**—This initially points to the internal function **qlinker**. It may some time in the future point to the function **foo**, if certain conditions are satisfied. (See later for more on this.)
- **Function name**—This is a pointer to the symbol **foo**. This is used by **qlinker**.



When a call is made to the function *foo*, the call actually is made to the address in the transfer table entry and ends up in the **qlinker** function. **qlinker** determines that *foo* is the function being called by locating the function name entry in the transfer table.<sup>1</sup> If the function being called is not compiled, then **qlinker** just calls **funcall** to perform the function call. If *foo* is compiled and if (status translink) is non-*nil*, then **qlinker** modifies the function address part of the transfer table to point directly to the function *foo*. Finally, **qlinker** calls *foo* directly. The next time a call is made to *foo* the call goes directly to *foo* and not through **qlinker**. This results in a substantial speed-up in compiled code to compiled code transfers. A disadvantage is that no debugging information is left on the stack, so **showstack** and **backtrace** are useless. Another disadvantage is that if you redefine a compiled function either through loading in a new version, or interactively defining it, then the old version may still be called from compiled code, if the fast linking described earlier has already been done. The solution to these problems is to use (sstatus translink value).

If (status translink) is *nil* all transfer tables are cleared. That is, all function addresses are set to point to **qlinker**. This means that the next time a function is called **qlinker** is called and looks at the current definition. Also, no fast links are set up. The result is that **showstack** and **backtrace** work and the function definition at the time of call is always used.

If the value of (status translink) is *on* the LISP system goes through all transfer tables and sets up fast links wherever possible. This is normally used after you have **fasted** in all of your files. Furthermore, **qlinker** makes new fast links if the situation arises, which is not likely unless you **fast** in another file.

If the value of (status translink) is *t* or any other value not previously mentioned fast links are made by **qlinker** if the called function is compiled.

## 12.9 Fixnum functions

The compiler generates in-line arithmetic code for *fixnum*-only functions. Such functions include **+**, **-**, **\***, **/**, **\**, **1+** and **1-**. The code generated is much faster than using **add**, **difference**, etc. However it only works if the arguments to and results of the functions are *fixnums*. No type checking is done.

<sup>1</sup>**qlinker** does this by tracing back the execution stack until it finds the **call subroutine** machine instruction that called it. The address field of the this instruction contains the address of the transfer table entry.





## 13 The top level

- 13.1 Introduction 13-1
- 13.2 A top-level for debugging LISP programs 13-1
- 13.3 How to use tpl 13-1
  - 13.3.1 Stepper commands 13-5
- 13.4 Tpl special symbols 13-6
- 13.5 Sample sessions with tpl 13-7
  - 13.5.1 Using the stepper 13-10
- 13.6 The file subsystem 13-12
  - 13.6.1 Commands in the file package 13-12
  - 13.6.2 Backup variables in the file package 13-14
- 13.7 File subsystem implementation notes 13-14
  - 13.7.1 Filein notes 13-14
  - 13.7.2 Fileout notes 13-14





## 13 The top level

### 13.1 Introduction

**Tpl** is the default top-level “listener” for FRANZ LISP. This program reads input from the keyboard, evaluates the input, and prints the value(s) returned by the evaluation. While it is possible for a LISP system to provide just this bare *read-eval-print* loop and be quite useful, most users prefer a more “user-friendly” top-level.

Part of the attraction of LISP is that this or any other top-level interface to the user is easy to change for special uses. In many cases, serious application programs replace **tpl** with a different top-level. Several widely-used programs replace it with an algebraic infix parser; others use a natural language (English) parser, or a database command language. Since the source text for **tpl** is available as *tpl.l*, the code can be used by programmers as a basis for other top-level “listeners.”

### 13.2 A top-level for debugging LISP programs

The particular goal of *this* top-level listener is to provide a natural link to FRANZ LISP debugging facilities, and support the programmer with various mechanisms to keep track of command histories, simplify the setting and examination of debugging flags, etc. **Tpl** provides enhanced debugging facilities, history command substitution, a file package, frame evaluation, and LISP stack manipulating functions.

### 13.3 How to use **tpl**

If you start up the FRANZ LISP system as delivered, **tpl** is the program which reads your keyboard input and determines what is done with it. **Tpl** prints a prompt `=>` .

Any input that is valid use at any level in FRANZ LISP will have exactly the same meaning to **tpl**, with the exception that new lines beginning with a question mark `?` are interpreted as special commands to **tpl**, and not passed immediately to LISP for evaluation.

In the description of the **tpl** commands, the notation `[...]` indicates optional arguments, and the notation `[a / b]` means *a* or *b* or neither.



### ?debug

■ Toggles the debugging state. When one is in debug mode, one can use debugging commands such as ?zo, ?up, and ?dn.

### ?debug full

■ Like ?debug except that one enters a break loop before an **errset**.

### ?help [topic]

■ Prints the help text associated with a particular command within **tpl**. **topic** can be selected from one of the **tpl** keywords; if no argument is given, a list of the keywords and a brief summary of their meanings is printed. For example:

---

```
=> ?help history
?history, ?his - print list of commands previously executed
?his r         - print results too
nil
=> ?help ?
??            - redo last user command
?? n          - (for n>0) redo command #n (as printed by ?history)
?? -n         - (for n>0) redo n'th previous command (?? -1 == ??)
?? symb       - redo last with car == symb
?? symb *     - redo last with car == symb*
nil
```

---

To get information on a function known to the LISP system, use the **help** function. For example, to get information on **lexpr-funcall** type (**help lexpr-funcall**).

### ?? [location-specifier]

■ Finds a particular previous command line identified by the **location-specifier**, and reexecutes it as if it were retyped to **tpl** directly. If the *location-specifier* is a non-numeric symbol, **tpl** scans backward through the commands to find an expression whose **car** is equal to the symbol. For example, ?? **print** will repeat the last command beginning (**print ...**). It is not necessary to type the whole symbol: you can type an asterisk to match "the rest of the atom." For example, ?? **pr \*** will also find (**print ...**) unless some more recent command also begins with (**pr... ...**). If the **location-specifier** is a positive integer, the command line with that number is reexecuted. If **location-specifier** is a negative number **-n** then the *n*th previous command is redone. If **location-specifier** is not given, then the last command is redone. Thus ?? is equivalent to ?? -1.



**?his[tory] [r]**

- Prints the history list of recent LISP commands. You may set the variable *tpl-history-show* to alter the number of the most recent commands which are displayed. Invoking the *r* option will display the results of those commands.

**?re[set]**

- EQUIVALENT TO: the LISP command (reset).

**?tr [fn1 fn2 ...]**

- Traces functions *fn1*, *fn2*, . . . . While this will usually be a simple enumeration of functions to be traced, more options can be passed to the trace function by using (name option-list) expressions as in the normal trace package (e.g. *?tr (foo break)*). See Chapter 11, §11.1.1.

**?untr [fn1 fn2 ...]**

- Untraces the specified functions, or if given no argument, will untrace all traced functions.

**?state [sym1 val1 ...]**

- Prints/changes the state of *tpl* flags and variables. The variables listed by *?state* are the only ones which can be changed via this command. *sym1* is set to *val1*, etc.

**?prt**

- 'Pop and retry': does a *?pop*, followed by a retry of the command which caused the last break loop to be entered. This is one of the most useful *tpl* commands, since it resumes computation, probably after a fix-up, from the last error.

**?ld [file1 file2 ...]**

- Loads the given files, or re-loads the just previously loaded file if no arguments are supplied.

**?fast**

- Sets up LISP for fast execution by turning off debugging mode (*?debug off*), setting *translink* to *on*, and setting *displace-macros* to *t*. Debugging information will be lost when this mode is entered. These settings generally would be used during the running of compiled programs which are known to be correct and in which high-speed execution is important.



### ?pop

- Pops up one break level. If at the top level, it has no effect.

### ?ret [val]

- Returns val from this break level. If the argument is missing, *nil* is returned. The value is returned to the function which produced the error, allowing it to continue. The break must have originated from invoking the **break** function or from the signaling of a continuable error. The argument val is evaluated.

### ?zo

- Views (zooms in on) a portion of the LISP stack. You may use **?up** and **?dn** to move the pointer to the current stack frame. Prior to using this you should execute the **tpl** command **?debug** so that sufficient information is stored on the stack.

### ?dn [n]

- Without arguments, moves the current frame pointer down one level and executes a **?zo**. If n is given, it moves that number of frames down. The stack grows upward, so the oldest frames are on the bottom.

### ?up [n]

- Is the same as **?dn**, except the current frame pointer is moved in the up direction.

### ?ev symbol

- Determines the value of symbol in the context of the current stack frame, as if the frames above the current one had not yet been created.

### ?pp

- "Pretty prints" the current frame with neat indentation and without ellipses. Ordinarily this would be used after **?zo** is used to locate a frame of interest.

### EOF

- The end-of-file character (*control-D* on the Unix operating system, *control-Z* on VAX/VMS) pops up one break level if it is typed to **tpl** from a keyboard input stream. Depending upon the catching of signals, if it is typed to the top-level, it may be used to exit from LISP.



### 13.3.1 Stepper commands

One can use the following commands to enter and use stepping mode<sup>1</sup>

**?step** [*t* | *fn1 fn2 ...*]

- Initiates a mode of single-step execution of LISP. If *t* is the argument, this is done immediately. Otherwise stepping is initiated upon entry to any of the functions *fn1*, *fn2*, ...

**?soff**

- Turns stepping off.

**?sc** [*n*]

- When in stepping mode, steps *n* times, then enters a LISP (break). The argument *n* defaults to 1. If *n* is the symbol *inf*, then steps forever without breaking.

**?l** [*n*]

- When in stepping mode, backs out of the current structure *n* levels.

**?c** [*n*]

- When in stepping mode, advances the stepper to the *n*th s-expression.

**?p** [*n1 n2 part*]

- When in stepping mode, if no arguments are presented, the current form is printed in full; with *prinlevel* and *prinlength* set to *nil*. If *n1* and *n2* are given, the current form is printed with the *prinlength* and *prinlevel* set to them. If *part* is given, it should be a structure accessing function (usually, *car*, *cadr*, *cdr*, etc.) which determines which part is to be displayed.

**?P** [*n1 n2 part*]

- This command is just like **?p** except the previous *result* is printed.

**?ppp** [*part*]

- When in stepping mode, if no arguments are presented, the current form is pretty printed. If *part* is given, it should be a structure accessing function (usually, *car*, *cadr*, *cdr*, etc.) which determines which part is to be displayed.

<sup>1</sup>Stepper enhancements contributed by Anthony Albert from the University of California at Berkeley.



**?PPP [part]**

- This command is just like **?ppp** except the previous *result* is pretty printed.

**?m**

- When in stepping mode, macroexpands the current form.

**?sp [arg]**

- This command controls when to print forms and results while stepping. If *arg* is not present, only forms and results at a higher level than the current one are displayed. If *arg* is a non-nil symbol, all forms and results are displayed. If *arg* is a number, only forms and results at a level higher than *arg* + 1 levels below the current level are displayed.

**RETURN**

- When in stepping mode, equivalent to **?sc 1**.

**SPACE RETURN**

- When in stepping mode, equivalent to **?c 1**.

## 13.4 Tpl special symbols

The following are special symbols:

*user-top-level*

- may be bound to a function (i.e. a *lambda* expression, or more likely a symbol which is the name of a defined function), which will be evaluated instead of (tpl) as the *read-eval-print* loop.

*top-level-eofs*

- if bound to a *fixnum*, used as the number of end-of-files to read at the top-level *read-eval-print* loop before exiting. The default value is 5. It is still true that the top-level will exit if an end-of-file is read and the input device is not a tty.

*top-level-prompt*

- if bound to a non-nil *s-expression*, will be used as the top-level prompt.

*top-level-init*

- if bound to a function, will be used to initialize **tpl**. Normally, the *.lisprc* file is read, and the copyright notice and version number are printed. Using this variable is a convenient way of setting **tpl** options in a *.lisprc* initialization file.

*top-level-print*

- if bound to a function, will be used to print values returned by the *read-eval* part of the *read-eval-print* loop.

*tpl-number-prompt*

- if *t*, will cause *tpl* to print an index number with the "*=>*" prompt.

*tpl-prinlevel*

- the maximum nesting level to print lists. Beyond that point, lists are abbreviated to *&*.

*tpl-prinlength*

- the maximum length to print a list. Beyond that point, lists are abbreviated to *&*.

*tpl-history-show*

- the number of history items to show with the *?history* command.

*displace-macros*

- if *t*, then displace *macros* with their expansion. This will speed execution, occupy more space, and possibly interfere with debugging by replacing *macro* calls by possibly obscure expansions.

## 13.5 Sample sessions with *tpl*

---

```
; Sample session 1.
;
; First we load in a factorial function.
=> ?ld fact
;; Loading file "fact.l"
(fact)
; We 'pretty print' the fact function.
=> (pp fact)
(def fact
  (lambda (n)
    (cond ((= n 0) (bug))
          ((times n (fact (sub1 n))))))
  t
; We somehow fail to notice that there is a bug in
; (fact) when n equals 0 so we try it out.
=> (fact 10)
Error: eval: Undefined function bug
Form: (fact 10)
```



```
; We could use showstack or backtrace to find out
; what is wrong, but for this example we decide that we
; want to use the more powerful ?zo (zoom) function.
; In order to use ?zo, we have to be in debugging
; mode before the error occurs. Since we are not, we
; decide to turn on debugging and run the function again
; so it gets an error.
c{1} ?debug
Debug is on
t
; The ?prt function pops up a break level and retries
; the function that caused the error. The line just below
; which says '=> (fact 10)' was printed by tpl.
; It was not typed by the user. Tpl is showing the function
; it is retrying.
c{1} ?prt
=> (fact 10)
Error: eval: Undefined function  bug
Form: (fact 10)
; The error occurred again. Now that we are in debug mode,
; we can do a zoom.
c{1} ?zo
Should I re-calc the stack(y/n):y
*** top ***
// current \
(bug)
(cond ((= n 0) (bug)) ((times n &)))
(fact (sub1 n))
(times n (fact (sub1 n)))
nil
; It shows that the current frame is the top frame and
; that is the evaluation of (bug).
; We can ask what the value of n is at this point:
c{1} ?ev n
0
; It is pretty clear that the problem is that the bug
; function is undefined. Before we correct it, we show a
; bit more of tpl. Here we go down five frames:
c{1} ?dn 5
(cond ((= n 0) (bug)) ((times n &)))
(fact (sub1 n))
(times n (fact (sub1 n)))
(cond ((= n 0) (bug)) ((times n &)))
// current \
(fact (sub1 n))
(times n (fact (sub1 n)))
(cond ((= n 0) (bug)) ((times n &)))
(fact (sub1 n))
nil
; Now we inquire as to n's value at this point in the
; execution:
c{1} ?ev n
2
```

```

; Now let us fix the bug. The function (fact) could be
; edited by using editf (see chapter 16), or we can define
; (bug) as returning 1.
c{1} (defun bug () 1)
bug
; Now we pop and retry. Notice that we didn't have to move
; the current frame to the top.
c{1} ?prt
=> (fact 10)
3628800
; This time it works.

```

---

```

; Sample session 2.
;
; Things work slightly differently when (fact) is compiled.
;
=> ?ld fact
Fast loading file "fact.o"
(fact)
; We turn on debugging since we know that an error will occur.
=> ?debug
Debug is on
t
=> (fact 10)
Error: Undefined function called from compiled code  bug
Form: (fact 10)
; Look at the stack.
c{1} ?zo
Should I re-calc the stack(y/n):y
*** top ***
// current \
a:(fact (0))
a:(fact (1))
a:(fact (2))
a:(fact (3))
nil
; The call to (bug) isn't visible on the stack, since
; undefined functions are detected in compiled code in
; a different manner. Notice that the frames are preceded
; by 'a:' and the arguments look unusual. This is an 'apply'
; form, which you may think of as a shorthand for (apply
; 'fact '(2)). This is how frames showing calls from
; compiled code look.
c{1} (defun bug () 1)
bug
; Again we fix the bug and retry.
c{1} ?prt
=>(fact 10)
3628800

```



; And it works.

---

### 13.5.1 Using the stepper

---

```
=> ?ld demo.1
;; Loading file "demo.1"
(demo.1)
; Pretty print demo
=> (pp demo)
(def demo
  (lambda nil
    (getd 'demo)
    (progn 1 2 3 4 5 6 7 8 9 10 11 12 13)
    (if (eq (car '(a b c)) 'a) 'true 'false)))

t
; Turn on stepping
=> ?step t
t
; Run demo. Execution pauses before "eval"ing the top level form,
; (here it is "(demo)"). An "e:" is prepended to the form since this
; represents a call to Eval.
=> (demo)
e:(demo)
; Prompt is now 'sc{1}'. '?sc' is typed to advance one "step".
; [Note that usually <return> would be typed instead of '?sc' without
; an argument. Only with commands like '?sc 3' is it necessary to use
; the long form of the command.]
sc{1} ?sc
e:(getd 'demo)
; Now execution pauses before the first s-exp in the function demo. Once
; again we type '?sc'.
sc{1} ?sc
e:'demo
; Now we're down another level within the s-exp. Another '?sc'
; completes the current s-exp, "'demo'", and also the higher
; level s-exp, "(getd 'demo)".
; The results of both are given following an '=' which
; represents results.
; Notice also that the position of the e:'s and ='s
; match and that the positions
; move to the right as the level increases (gets deeper).
; This leaves execution paused before the second of
; the three s-exps in the function.
sc{1} ?sc
==demo
==(lambda nil (getd 'demo) (progn 1 2 3 ...) ...)
```



```

e:(progn 1 2 3 ...)
; We are interested in seeing the complete result of the
; last s-exp. The s-exp was "(getd 'demo)". To pretty print the
; whole form returned, we type '?ppp'.
sc{1} ?ppp
(lambda nil
  (getd 'demo)
  (progn 1 2 3 4 5 6 7 8 9 10 11 12 13)
  (if (eq (car '(a b c)) 'a) 'true 'false))

nil
; Now we want to see the complete form execution is paused at.
; To see this using the normal "print", we type '?p'.
sc{1} ?p
(progn 1 2 3 4 5 6 7 8 9 10 11 12 13)
nil
; Now we type '?sc' to begin evaluating the arguments to progn.
; Execution pauses at the first argument, "1".
sc{1} ?sc
e:1
; Another '?sc' gets us to "2", while returning the result from "1".
sc{1} ?sc
==1
e:2
; This time we skip the next 3 steps by doing a '?sc 3'.
; Execution is paused at "6". The result from "2" is displayed.
sc{1} ?sc 3
==2
e:6
; Now, instead of going through all 13 arguments,
; we type '?1' to back out one level. The result of "6" is displayed
; as is the result from the "progn" form.
; Execution is paused at the macro "if". Notice that an "m:" is
; prepended to this form, signifying "macro".
sc{1} ?1
==6
==13
m:(if (eq (car &) 'a) 'true 'false)
; The most convenient way to deal with macros when stepping
; is to expand them.
; We type '?m' to expand the macro. Execution is paused at
; the same form, at the same level, but with all macros at all
; levels within the form expanded.
sc{1} ?m
e:(cond ((eq & &) 'true) (t 'false))
; To see the entire s-exp we use '?ppp'.
sc{1} ?ppp
(cond ((eq (car '(a b c)) 'a) 'true) (t 'false))
nil
; We go forward one step using '?sc'.
sc{1} ?sc
e:(eq (car '&) 'a)
; We can step over the whole s-exp here with '?c'. The result, "t", is

```



```
; displayed. Execution pauses at the next step after this s-exp.
; [Note that like '?sc', there is a short form
; for '?c', <space> <return>
; (that is, a space followed by a carriage return).]
sc(1) ?c
    ==t
    e:'true
sc(1) ?l
    ==true
    ==true
    ==true
true
=>
```

---

## 13.6 The file subsystem

The FRANZ LISP file package helps support the residential environmental style of LISP programming in which most or all program editing is done within LISP itself, probably using **editf**, **editv**, and **editp** to create and debug programs. Although ordinary data files are used by the file package to store the programmer's alterations to the function definitions and other data that persist between runnings of programs, management of those files is controlled by the FRANZ LISP system.

As an interactive session proceeds, the file package (in cooperation with the top-level) tracks the changes the user makes to the LISP environment. Those changes are of three types: function (or *macro*) definitions, values of variables (symbols) altered, and properties of symbols altered. At any point the user can find out what has been changed by typing **?changed** to the top-level, which will print the information out in a table form.

Each item (function, value or property) may be associated with a file. The list printed by **?changed** will show the associated file for each changed item. In order to save a change, the user must request that the associated file be written out (using **?fileout**, described below). If an item doesn't have an associated file, then one can be declared using **?add-function**, **?add-var**, or **?add-prop**, depending on the type of item.

### 13.6.1 Commands in the file package

**?filein** [name1 name2 ...]

- loads the named files using a read-eval loop, printing the names (not the values) being loaded. The files being read should have been written with **?fileout**. If no files are named as arguments, a list of all previously loaded files is returned.



**?fileout [name1 name2 ...]**

- writes the given files if any of the items in the file have changed. With no arguments, all files that need updating are rewritten.

**?changed**

- Reports on those data items which have changed but not stored on files, and the names of associated files.

**?add-function filen fcn1 [fcn2 ...]**

- Adds fcn1, fcn2, etc. to the list of functions associated with the file filen. The file filen should either not exist or should be the name of a file which has been loaded with **?filein**.

**?add-var filen var1 [var2 ...]**

- Adds the given symbols var1, var2, etc. to the list of symbols stored in the file filen. The file filen should either not exist or should be the name of a file which has been loaded with **?filein**.

**?add-prop filen (sym1 ind1) [(sym2 ind2) ...]**

- Adds symi's indicator property to the list of properties stored in the file filen. The file filen should either not exist or should be the name of a file which has been loaded with **?filein**.

**?rem-function filen fcn1 [fcn2 ...]****?rem-var filen var1 [var2 ...]****?rem-prop filen (sym1 ind1) [(sym2 ind2) ...]**

- Remove the named items from filen. They do this by deleting the association of the item from the file. When the file is next written with **?fileout**, the items will not be written out.

**?whichfile fcn | var | (symbol ind) ...**

- For each item (which can be a function, variable or (symbol ind)), prints the associated filename, if there is one. If a symbol is both a function and a variable (in different files), both associated files are printed.

**?filestatus [filen1 filen2 ...]**

- Prints the names of the items in each file listed. If no filenames are given, prints a summary status report of all files.



### 13.6.2 Backup variables in the file package

There are several variables which the user might wish to alter to assist in backup maintenance:

`file-backup-prepend`

- A string (or symbol) to prepend to the filename to generate a backup filename during a `?fileout`.

`file-backup-append`

- A string (or symbol) to append to the filename to generate a backup filename during a `?fileout`.

### 13.7 File subsystem implementation notes

The file package maintains a database of knowledge about files. For each file it keeps track of the items stored in that file. The file package also maintains a list of items which have changed, called the `changed-list`.

#### 13.7.1 Filein notes

`filein` recognizes three types of items: functions, variables and properties.

A **function** item has this form: (kwd functionname anything ...) where kwd is an element of the list which is the value of `file-function-modifiers`. The initial value of `file-function-modifiers` is (defun def defmacro). The user may wish to add something to this list to read in a file not created by `?fileout`. The `?fileout` function-printing function will only use the **def** form, which provides a superset of the capabilities of the other forms.

A **variable** item has this form: (kwd variablename anything ...) where kwd is an element of the list which is the value of `file-variable-modifiers`. The initial value of `file-variable-modifiers` is (setq).

A **property** item has this form: (kwd symbol anything indicator) where kwd is an element of the list which is the value of `file-property-modifiers`. The initial value of `file-property-modifiers` is (defprop). Note that the symbol and indicator are not evaluated before they are added to the list of items, so **putprop** is not a valid kwd to be added to `file-property-modifiers`.

#### 13.7.2 Fileout notes

Files created by `?fileout` contain only a few types of forms (**def**, **setq** and **defprop**). If the file is edited externally from the LISP system and other forms are inserted (such as **declares** or comments), and then the file is filed in-and-out, the other forms will be lost. It is also important to keep forms syntactically correct (e.g. with parentheses balanced), because then forms following the error will not be read in to the LISP system. It is generally safe to edit or merge files to add, delete or alter syntactically proper



definitions of the forms already known to the file package.

**?fileout** performs the following sequence of operations: it opens up a file in */tmp* and writes all items in the file. As each item is written, it is also removed from the global changed-list if it was on that list. If a file with the same name as the one being written exists then **?fileout** will preserve the previous file by changing its name, if the user has set one or both of the variables *file-backup-prepend* and *file-backup-append*. If both of these variables are *nil*, then a backup will not be done. If *file-backup-prepend* is non-*nil*, then its value should be a symbol or string which will be prepended to the filename in order to create the backup name. Likewise *file-backup-append* will be appended to the filename to create the backup name. If both variables are non-*nil*, then both will be used. Finally, the file in */tmp* is renamed to the name of the file being filed out.

A caution is appropriate: suppose you start LISP and define the function *solveit*. You would like to add this function to the file *eqn.l* which you created earlier and which already contains a number of functions. Your first thought may be to type:

```
?add-function eqn.l solveit
```

Since the file *eqn.l* exists on the disk but hasn't been loaded yet, the file package is ignorant of any functions other than *solveit* associated with *eqn.l*. Executing

```
?fileout eqn.l
```

would cause the contents of *eqn.l* to be replaced with the definition of the single function *solveit*. As a guard against this situation, the file package asks you if you want to abort the **?add-function** operation when you mention an existing file which however has not been read-in. It is best to type "yes" at this point, then

```
?filein eqn.l
```

and then

```
?add-function eqn.l solveit
```



The first part of the document discusses the importance of maintaining accurate records of all transactions. It emphasizes that proper record-keeping is essential for the integrity of the financial system and for the ability to detect and prevent fraud. The document also outlines the specific requirements for record-keeping, including the need to maintain records for a minimum of seven years and to ensure that all records are properly indexed and filed.

In addition, the document discusses the importance of regular audits and the role of the auditor in ensuring the accuracy of the records. It states that audits should be conducted at least once a year and that the auditor should report any discrepancies or irregularities to the appropriate authorities.

The document also discusses the importance of training and education for all personnel involved in the financial system. It states that all personnel should receive regular training and education to ensure that they are up-to-date on the latest developments in the field and that they are able to perform their duties effectively and efficiently.

Finally, the document discusses the importance of transparency and accountability in the financial system. It states that all transactions should be properly documented and that all personnel should be held accountable for their actions. This is essential for the public's confidence in the financial system and for the overall health of the economy.

## 14 Structured data types

- 14.1 Introduction 14-1
- 14.2 Setf and defsetf 14-1
- 14.3 Defstruct 14-4
  - 14.3.1 Defstruct argument forms 14-5
    - 14.3.1.1 Using defstruct 14-13
      - 14.3.1.1.1 Types of structures 14-13
      - 14.3.1.1.2 Constructor functions 14-14
      - 14.3.1.1.3 Alterant macros 14-15
      - 14.3.1.1.4 Other caveats 14-16





## 14 Structured data types

### 14.1 Introduction

Chapter 2 described the basic data types of FRANZ LISP objects and the functions which access and manipulate them. In most advanced LISP applications, however, these simple types are used to create more complex forms which may combine or even nest these data types to form a **structure**. There are no built-in functions for manipulating these user-defined structures; they must be created by the user. However, this implementation of FRANZ LISP contains a number of facilities for assisting the user in the creation and maintenance of structures.

### 14.2 Setf and defsetf

Consider, for a moment the list (call it *alist*):

(foo bar).

We refer to foo as the **car** of the list; bar is the **cadr**. The LISP function which returns the **car** of a list is the *function car*. In the above example, foo can be thought of either as the result of evaluating (car alist) or as *the car of alist* where *car of alist* is used as another **name** for foo.

Another way of looking at this is to think of each LISP data type as having a function which *accesses* it and a function which **modifies** it. However, the **function** which accesses a data type can also be thought of as the **name** of object which is accessed. The simplest case of this occurs in variable binding. For example:

(setq var 'value)

assigns the value of var to *point to* value. The **access** form is var itself, since typing var to the top-level will cause it to return 'value. The name of the variable is the form by which it is accessed. It would be useful to be able to eliminate the need for a separate modifier function by using the access function to **modify** the data type as well. The function **setf** accomplishes this.

This chapter to the FRANZ LISP manual describes additions by contributors from the Massachusetts Institute of Technology, the University of California at Berkeley, and the University of Pittsburgh.



(setf g\_accessfn1 'g\_val1)

■ WHERE `g_accessfn` is the complete function call which would access the desired data and `g_val` is the value to which it will be assigned.

■ SIDE EFFECT: **setf** is a *macro* which replaces itself with a call to the proper update function.

■ RETURNS whatever the value of the *macro* expansion returns.

□ In the simplest case, then, **setf** can be used to replace the function **setq**. But consider, for a moment, some other data types. Returning to the example of the list, above, we referred to the first element of the list as the **car** of the list. The function which would update this value is the function **rplaca**. But with **setf** we could use **car**, instead.

---

; we can use setf as we would setq:

```
=> (setf alist '(foo bar))  
(foo bar)
```

; now, to replace the car of alist:

```
=> (setf (car alist) 'baz)  
(baz bar)
```

; is it a miracle?

; no, not really:

```
=> (macroexpand '(setf (car alist) 'baz))  
(rplaca alist 'baz)
```

---

□ Lest one think that **setf** is all done with magic, it should be pointed out that the objects which can be updated by **setf** must be predefined using the *macro* **defsetf**. For instance, to make the access function **car** known to **setf**, we must define it:

```
(defsetf car (e v) '(rplaca ,(cadr e) ,v))
```

(defsetf s\_fname l\_setfvars 'g\_body)

■ RETURNS a *lambda* form which describes the expansion of the **setf**.

■ WHERE `s_fname` is the name of the access function, `l_setfvars` is a list of the arguments to the **setf** call, and `g_body` which is the body of the *lambda* expression to be created.

■ SIDE EFFECT: **defsetf** is a *macro* whose argument list is in the same form as **defun**.<sup>1</sup> The only substantial difference between **defsetf** and **defun** is that in the case

<sup>1</sup>The defining forms of a *lambda* expression are described in Chapter 8 in the section on **defun**. Since **defsetf** is, essentially, a **defun**, all of the **defun** defining forms will work for **defsetf**.



of **defsetf** the *lambda* form is not defined at the top-level,<sup>2</sup> but rather as the value of the property **self-expand** on the the property list of *s\_fname*.

□ In order to better understand the workings of **defsetf**, it is useful to examine the implementation of **setf**. The following example illustrates some of the *inner workings* of the **setf** evaluation:

---

```
; as in the example, above

=> (defsetf car (arg1 arg2) `(rplaca ,(cadr arg1) ,arg2))
(lambda (arg1 arg2) (list 'rplaca (cadr arg1) arg2))

; now, if we examine the property list of car:
=> (pp-form (plist 'car))
(setf-expand (lambda (arg1 arg2)
              `(rplaca ,(cadr arg1) ,arg2)))
nil
```

---

■ In evaluating (**setf** *g\_accessfn* '*g\_val*) the following algorithm is followed:

- 1 If *g\_accessfn* is atomic and a symbol, then *g\_accessfn* is **setq**ed to *g\_val*.
- 2 If *g\_accessfn* is non-atomic and the **car** of *g\_accessfn* is a symbol, then the property list of *g\_accessfn* is searched for the indicator **self-expand** and the property value (presumably a *lambda* expression created by **defsetf**) is then **apply**'d to *g\_val*.
- 3 If *g\_accessfn* is non-atomic and the **car** of *g\_accessfn* is a symbol representing some combination of **car** and **cdr**, then an update function is created and applied to *g\_val*.<sup>3</sup>
- 4 If *g\_accessfn* is non-atomic and the **car** of *g\_accessfn* is a *cmacro* or *macro*, then the *macro* is expanded and **setf** reapplied to the result.
- 5 If *g\_accessfn* is non-atomic and the **car** of *g\_accessfn* is a function, then **setf** is reapplied using the function binding of the **car** of *g\_accessfn*.
- 6 If no other condition is satisfies, **setf** *g\_accessfn* is continually reapplied (or expanded), until either **setf** quits or the situation is resolved.

<sup>2</sup>Meaning the *lambda* expression created by (**defsetf** *s\_fname* | *vars* '*g\_body*) will not become the function binding of *s\_fname*.

<sup>3</sup>By implication (unless otherwise specified), there is no property **self-expand** for most of the **car**, **cdr** combinations as these are *created* on the fly.



setf form	update form	object type
(setf (arg argnum) newval)	(setarg argnum newval)	nlambda
(setf (arraycall type arr key) newval)	(store arexp newval)	array
(setf (cxr key obj) newval)	(rplacx key obj newval)	hunk
(setf (get obj key) newval)	(putprop obj newval key)	property list
(setf (gethash obj key) newval)	(puthash obj newval key)	hash table
(setf (nth keyex list) newval)	(rplaca (nthcdr keyex list) newval)	list
(setf (nthcdr keyex list) newval)	(rplacd (nthcdr (sub1 keyex) list) newval)	list
(setf (nthelem keyex list) newval)	(rplaca (nthcdr (sub1 keyex) list) newval)	list
(setf (plist obj) newval)	(setplist obj newval)	property list
(setf (symeval obj) newval)	(set obj newval)	symbol
(setf (vref obj key) newval)	(vset obj key newval)	vector

□ For convenience, a number of **setf macros** have already been defined in FRANZ LISP. These are shown in the above table. In addition, **setf** can be used to update structures defined by **defstruct**.

### 14.3 Defstruct

The notion of using an access function as an update function (as manifested in the **setf macro**) can be carried even further. It is not hard to imagine, for example, that the user may want to *nest* lists within lists and data forms within data forms to create more complex data structures. In such a complex structure it might be useful to name *components*.<sup>4</sup> to provide an easy way to access the imbedded data forms. Furthermore, access and update functions would have to be created for each element of that structure.

Consider the following example: Suppose we wished to write a FRANZ LISP program that dealt with ships. Insofar as we are concerned, the relevant data regarding each ship is its *x* and *y* position in a two-dimensional coordinate system; its velocity in each coordinate, and its total mass. We could represent the information regarding this ship as a list of five elements, an array, a vector, or any other combination of LISP data types. But how easy would it be to manipulate this data? In reading over our programs would we remember that (caddr ship) was the *velocity in direction y* of ship?

Instead, what we would like to find some way of representing ship so that the names of the components of the ship provided a way to access and update ship information. That facility is provided by **defstruct**.

```
(defstruct (ship :conc-name)
  x-position
  y-position)
```

<sup>4</sup>The notion of *structures* such as those we will be talking about has been described for many languages (in one variety or another), so that it becomes impossible to find *neutral* terms to describe the form of these structures. For example: the term *record* has meaning to both Pascal and InterLISP users, though it would not describe exactly the same thing to both. Similarly, we will refer to *structures* in FRANZ LISP although the name is familiar to PL/1 and 'C' users, as well. In the present case, the term *components* is used to describe what might also be called *fields*, *elements*, or *slots*, principally because that term has precedent in other texts (e.g. *Weinreb and Moon* and *Steele*). The user is cautioned to consider these structures in FRANZ LISP as being *similar to* rather than *equivalent to* structures in other languages.



x-velocity  
y-velocity  
mass)

says that every structure *ship* has five components. For each of these five components there will be defined an *accessor function* which will return the value of that particular *component* which it names. For example, for any object (say QE2), which is an instance of *ship*, the value of x-position for QE2 will be:

(ship-x-position QE2)

Furthermore, **defstruct** will define a *macro*, **make-ship**, which can be used to create new instances of *ship*. This *macro* is known as a *constructor macro* since it constructs new structures of type *ship*.

Finally, as we might have expected from the previous section, structures defined by **defstruct** can be updated using the **setf** *macro*.

(setf (ship-x-position QE2) 50)

will set the value of the x-position component of QE2 to be 50.

(defstruct sl\_nameargs sl\_slotdescript1 [...sl\_slotdescripti])

- RETURNS the name of the structure created.
- SIDE EFFECT: creates a structure and the corresponding accessor and constructor functions for that structure.
- WHERE the arguments to **defstruct** correspond to the forms illustrated in the next section.

### 14.3.1 Defstruct argument forms

The slot description arguments *sl\_slotdescriptn* consist of simply a name or a list containing a name and an initial value.

In the simplest case, *sl\_nameargs* can be a symbol which will be the *name* of the structure created. This was illustrated by the case of the *ship* above. In a more complex case, *sl\_namelist* can be a list containing the structure name and a list of keywords. In most cases, the keywords may have an argument. When a keyword-argument pair is given, it should be in the form (keyword argument), unless indicated, otherwise. In FRANZ LISP, the argument is not evaluated. The following keywords have been defined for FRANZ LISP.

**:conc-name** [prefix]

- This keyword argument can have two forms, both of which provide automatic prefixing of access function names. When **:conc-name** is not given an argument, the name of the structure is concatenated to the component name with an intervening hyphen. For example:

(defstruct (person :conc-name) name age sex)



will create a structure *person*, with the *slots* *name*, *age*, and *sex*, and the *access functions* named **person-name**, **person-age**, and **person-sex**. On the other hand, when given an argument, expressions like:

```
(defstruct (person (:conc-name human)) name age sex)
```

will create a structure *person* with the access functions **humannname**, **humanage**, and **humansex**. (Notice the absence of the hyphen in this case.)

### **:named**

■ Means that, where possible, the structure created will be a *named* type (i.e., if the type would have been a *:list*, it will be a *:named-list*, instead). In addition, certain functions exist which operate on named structures.

**:include** structure [component | (component initial-value) ...]

■ The **:include** option can be used to build new structures which are composites of other structures. The structure so defined will have the all of the slots of the **:included** structure in addition to to the newly defined slots. Access functions which are defined for the **:included** structure will operate for the structure in which it is included but the converse is not true.<sup>5</sup>

□ Consider the following:

---

```
; suppose we wish to create a structure, astronaut.
; since astronauts are people, too
=> (defstruct (astronaut :conc-name (:include person))
    helmet-size
    (favorite-drink 'Tang))
astronaut

; now, to create an instance of astronaut
=> (setq Glenn (make-astronaut
    name 'John
    age 45
    sex 'male
    helmet-size 17.5))

#<astronaut 5>

=> (person-name Glenn)
John

; since person-name was inherited from the :included file
```

<sup>5</sup>This is also the case in ZetaLISP, but not the case in Common LISP, that is, in Common LISP, the structure *astronaut* which **:included** the structure *person* would have two access functions for the slots in *astronaut* which were also in *person*. That is to say, for *astronaut* as defined in the next example, there would be two accessor functions for the *name* slot: **person-name** and **astronaut-name**. This may, someday, be the case for FRANZ LISP as well.



```
; astronaut-name is not defined
=> (astronaut-name Glenn)
Error: eval: Undefined function astronaut-name

=> (astronaut-helmet-size Glenn)
17.5
```

---

Notice that **:conc-name** in the structure was only passed to the structures uniquely defined within the **defstruct**. Those components which were **:included** inherited their accessor function names from their parent structures. In addition, if default values exist for the structure which is **:included**, these values are inherited by the new structure unless an alternate form is used, such as:

```
(defstruct (astronaut (:include person sex (age 45)
                    :conc-name)
            helmet-size
            (favorite-drink 'Tang))
```

which says that the default value for **sex** is *nil* and for **age** is **45**. As we can see from this example, the proper way to specify defaults values is either as a *list* whose **car** is the slot and whose **cadr** is the value for that slot or as a *symbol*, where the value defaults to *nil*.

□ Multiple **:includes** cannot be used. For example, suppose that structure A has 5 slots and structure B has three, the result of:

```
(defstruct (C (:include A) (:include B))
  slot1c slot2c slot3c)
```

will be a structure whose first three slots come from B and whose last three slots are from C; no slots from A will be used. This *feature* is present because of the fact that the slots of a structure are allocated starting from the beginning of that structure.<sup>6</sup> Since the accessor functions of B cannot be changed by their subsequent use in another structure, the reference point for accessor functions is almost always the first element of the type in which that structure was implemented. If a second structure were included, it would, by necessity, be forced to start at the first slot of the new structure since the accessor functions were previously defined. For this reason, only the last structure is **:included** if **:include** is used more than once. If this is not entirely clear, read through the next section before coming back to this point.

□ In FRANZ LISP **:included** structures must be of the same **:type** as the aggregate structure which **:includes** them. This is further explained in the next description

<sup>6</sup>Hence, the first slot in a structure of **:type :list** will be the **car** of the list.



**:type** type-name

■ The **:type** option specifies which type of LISP object will be used to implement the structure. In FRANZ LISP the default type is a *named-vector*. As mentioned, above, the **:type** of an aggregate structure cannot be different from the **:type** of the structures which are **:included** in it. By implication, this means that structures cannot be composed of other structures of more than one type. The reason for this is clear: the accessor functions are defined at the time that the structure is created. Since these accessor functions are type dependent one cannot create a structure which would alter the type of a parent structure.

**:named**

■ Structures which are **:named** are implemented as *named types*; by default, *named-vectors*. As always, aggregate structures can only **:include** structures of the same type, therefore, in the previous example, if *astronaut* is named so must be *person*. The types available in FRANZ LISP are described in the next section.

**:constructor** [name [init-list]]

■ This option may have, zero, one or two options. If the argument is not given or if **:constructor** is not given as an option to **defstruct**, then a *keyword driven* constructor is defined whose name is the symbol *make-* concatenated to the name of the structure defined (e.g., **make-person** in the case of *person*). If one argument is given and is *nil*, no constructor will be defined. If one non-*nil* argument is given then the *keyword driven* constructor *macro* is named with that argument. If two arguments are given, the first must be a symbol (the name of the constructor), and the second a list of the slots initialized by that *macro*. In this last case, any initializations which are done in the constructor will override the initializations defined in the body of the **defstruct**. In contrast to the previous cases, **:constructor** with two arguments is *by position* rather than *keyword-driven*.

□ The use of **:constructor** will be illustrated in the next section.

**:default-pointer**

■ In the definition of **defstruct** we outlined the procedure for assigning *default values* to slots of the structure being defined. In other applications, we may wish to create a structure which, apart from the structure definition, can serve as a model or default case of that structure. One way to do this is through the **:default-pointer** option.

□ Under normal conditions, accessor functions such as **person-name** take one argument, the name of the instance we wish to reference. By using **:default-pointer**, the argument to the accessor function becomes optional. If it is not specified, the value returned by the accessor function is the value of the slot for the instance pointed to by the argument to **:default-pointer**. To illustrate:



---

```
=> (defstruct (apple :conc-name
                    (:default-pointer *APPLE*))
      (type 'fruit) name color)

apple
=> (setq *APPLE* (make-apple color 'red))
#<apple 3>
=> (setq crabapple (make-apple color 'green name 'crab))
#<apple 3>

; by default, the accessor functions will point to *APPLE*:
=> (apple-color)
red
=> (apple-color crabapple)
green
```

---

**:size-symbol** name

■ This option allows a user to specify a global variable which will be bound to the *size*<sup>7</sup> of the structure being created. If no option is given the variable will be called *struct-size* where *struct* is the name of the structure. Otherwise the variable will be named by the argument to **:size-symbol**.

**:size-macro** name

■ This option is, in every respect, equivalent to **:size-symbol** except that rather than creating a symbol, this option creates a *macro* which *expands* to the value of the size of the structure.

**:initial-offset** offset

■ This option **requires** one argument (a *fixnum*), which will tell **defstruct** to skip some number of slots before assigning the slot names. No accessor functions are created for these slots, so the use of this option presupposes some knowledge about the implementation of **defstruct** on the part of the user. If the structure **:includes** another structure, then the order of slot assignments is: **:include**, **:initial-offset**, then **defstruct**.

**:callable-accessors** [arg]

<sup>7</sup>By *size* we are referring to the the number of *elements* at the top-level of the structure, therefore, the *size* of the list (a (b c) d (e f g)) is 4.



■ Normally in FRANZ LISP, accessor functions are not functions but *macros*.<sup>8</sup> If the option **:callable-accessors** appears in the **defstruct** with an argument of *t* or no argument at all, the accessor functions are defined as *lambda* functions, rather than as *macros*.

---

N.B. While this option provides accessors which can be used as conveniently as functions (i.e., they can be passed as arguments to **mapcar**), the accessor functions created cannot be used to update slots using **setf**.

---

#### **:eval-when** when-keywords

■ This option behaves just as the compiler function **eval-when** and has the same arguments only in keyword form (preceded by a colon. **:**). By default structure-associated functions are:

(:eval-when (:eval :compile :load))

#### **:alterant** [symbol | nil]

■ Normally when a structure is created a function is also defined which can be used to alter, *en masse* all of the slots in a particular instance of that structure. The name of this function<sup>9</sup> is made by concatenating *alter-* to the structure name, (e.g. **alter-person** in the example above). The argument to **:alterant** should be either a symbol (which will be the name of the alterant instead of the default), or *nil*, which means no alterant function will be defined. Using **:alterant** without an argument is equivalent to not using it at all; in these cases the default function will be defined.

□ Alterant functions are described in the next section.

#### **:but-first**

■ In some instances, a structure will be defined which will **always** be part of another structure, so that the accessors of that structure must reference the structure of which it is a part. The **:but-first** option is a means of expressing this relationship. Consider the following example:

---

```
; using person as an example, we may wish to subdivide
; name into three more slots (of type :list)

=> (defstruct (name :list (:but-first person-name))
      firstname middlename lastname)
name
```

<sup>8</sup>In contrast to accessor functions in ZetaLISP which are, by default, *subsrs* (functions).

<sup>9</sup>Actually, a *macro*.



```

; and to create an instance of person called generic-person:
; (with a name more colorful than "John Q. Public")

=> (setq generic-person (make-person name (make-name)))
#<person 3>

=> (alter-name generic-person firstname 'Rufus
      middlename 'Thaddeus lastname 'Firefly)
(Firefly)

; to alter the head field of body

=> (setf (middlename generic-person) 'T.)
(T. Firefly)
=> (person-name generic-person)
(Rufus T. Firefly)

```

---

The **:but-first** option is used to create accessor functions which apply their arguments to the result of another accessor function which was the argument to **:but-first**.<sup>10</sup> This option always takes one argument, and always defines a structure which will exist only within the context of another structure.

□ Notice how an instance of person is created in the example, above. The call to **make-person** includes the assignment of the name slot to a call to **make-name**. This procedure must be followed when using **:but-first** structures: the substructures must be created (using the **make-** functions), at the same time that the top-level structure is instantiated. To see what would happen if this procedure is not followed, imagine the case if one had defined **test-person** without including **make-name**:

---

```

=> (setq test-person (make-person))
#<person 3>

; note that there will be exactly one
; slot for the value of person-name

; since alter-name expects person-name to
; be a three element list:
=> (alter-name test-person
      firstname 'John middlename 'Q lastname 'Public)
Error: Attempt to rplac[ad] nil.
<1>:

```

---

<sup>10</sup>In fact, the argument to **:but-first** should always be the name of an accessor function.



**:print-function** u\_func

■ u\_func should be a function of three arguments that can be used to print structures of the type being declared in the **defstruct**.

**:print** control-string [control-args ...]

■ Structures which are of a type **:named-vector** are allowed an addition option not available to other **:types**, the **:print** option. Basically, **:print** allows the user to define a format for printing the structure, using the directives defined for the **format** output package (see Chapter 5 for further details). The form of the arguments to **:print** is:

(:print control-string &rest args)

where args can reference any of the slots within the structure. For example, using a modified form of our ship structure:

---

```
=> (defstruct
      (ship (:print
              "~&A ship at position <~s,~s>~& with velocity [~s,~s]~&"
              (ship-x-position ship) (ship-y-position ship)
              (ship-x-velocity ship) (ship-y-velocity ship)))
      (ship-x-position 0)
      (ship-y-position 0)
      (ship-x-velocity 0)
      (ship-y-velocity 0)
      ship-mass)
      ship

; now, using make-ship, note the form of the output
=> (setq x (make-ship ship-x-position 10))
A ship at position <10,0>
  with velocity [0,0]

; if we modify some fields:
=> (setf (ship-y-velocity x) 25)
25
=> (setf (ship-x-velocity x) 'unknown)
unknown

; this time:
=> x
A ship at position <10,0>
  with velocity [unknown,25]
```

---



### 14.3.1.1 Using defstruct

#### 14.3.1.1.1 Types of structures

As was mentioned, previously, by default, structures are implemented as *named-vectors*. In the case of vectors, the property of the vector becomes the **name** of the structure.<sup>11</sup>

The following types are available in FRANZ LISP. The user writing portable code should be cautious in specifying only those types common to all LISPs of interest.

##### **:list**

- Implements the structure as a simple list.

##### **:named-list**

- Like **:list** but the first element of the list will be the name of the structure (therefore the length of a **:named-list** implementation of a structure will be one greater than the same structure implemented as a simple **:list**).

##### **:list\***

- Like **:list**, but the last **cons** of the list created is a *dotted pair*. There is no **:named-list\***.

##### **:array**

- Implements the structure as an array.

##### **:named-array**

- Like **:array** but the first element of the array will be the name of the structure. This type may or may not be available in your FRANZ LISP implementation.

##### **:hunk**

##### **:named-hunk**

- At one time **:hunk** was the default in FRANZ LISP. Its use, along with the use of **:named-hunk**, is now discouraged. Use vectors instead. The *zeroth* element of the *hunk* is the name of the structure when **:named-hunk** is specified.

##### **:tree**

- This structure is implemented as a binary tree of *cons cells* with the leaves serving as the slots. The advantage of using a tree as opposed to a list is that the *access time* for any slot at some level of branching is the same as the access time for any other

<sup>11</sup>If the structure is implemented as a list, the **name** is the **car**; if the structure is an array, the **name** is the first element, and so on.



slot at that same level. A disadvantage of using **:tree** structures is that they may not include or be included by any other structure.

**:fixnum**

■ A *fixnum* structure is a structure of one slot which is implemented as a single *fixnum* which, itself, may contain a number of slots, corresponding to the bits which make up the *fixnum*. These can be particularly useful in those LISPs where *fixnums* are implemented in byte sizes of 15 or 20 since many commonly used numbers can be represented in less than 10 bytes and, therefore, more than one number can be packed into a single *fixnum* space. In FRANZ LISP *fixnums* are only 10 bits and the utility of this type is questionable.

**:vector**

**:named-vector**

■ Structures are, by default, **named-vectors**. A named vector is called thus because it prints as **#<structure-name size>**. Vectors are described in Chapter 9.

### 14.3.1.1.2 Constructor functions

In the previous section we described a way to define the constructor function for a structure. This section describes the uses of **:constructor**.

When **defstruct** is applied without the **:constructor** option, a constructor function is automatically defined and named as the concatenation of *make-* to the structure name, (e.g, **make-person** in the case of the structure **person**). The constructor defined is a *keyword driven* function, that is, arguments to the function are preceeded by *keywords* which indicate which slots to fill. For example:

```
(setq fred (make-person name 'Fred age 22 sex 'male))
```

creates an instance of **person**, with the slots initialized according to the given keywords, *name*, *age*, and *sex*. In contrast, **:constructor** functions which are explicitly defined are *by-position* constructor functions, that is, values are assigned to the individual slots on the basis of their position in the function call. This is analogous to the scheme used to apply *lambda* functions in which *lambda* variables are assigned on the basis of the the position of the arguments to the function.

When explicitly defined, constructor provide a means for creating instantiations of a particular structure which can be tailored to the individual application. For example, in the case of the structure **person**, we know from experience that this category can be further subdivided according to certain criteria, some of which were used as *slots* in the definition of **person**. On the basis of *age* and *sex*, we can further subdivide **person** into *male*, *female*, *adult*, and *minor*.

Consider, as an example, the following expanded version of **person**:

```
(defstruct (person :conc-name :named :constructor
  (:constructor create-female (&aux (sex 'female))))
```



```
(:constructor create-male (&optional name age &aux (sex 'male)))
(:constructor create-adult (&optional sex &aux (age '>21)))
name age (sex 'unknown))
```

This says to define the named structure **person**, and in addition, define three functions for constructing certain instantiations of **person**: **create-male**, **create-female**, and **create-adult**.<sup>12</sup> The default sex for a person so created is *'unknown*. In addition, specific kinds of persons can be instantiated using one of the constructors. These have been defined, arbitrarily, to indicate the various ways that an argument list might appear; the argument list corresponds to a *lambda* list with a few minor variations:

- **&optional** arguments are given the value of the calling arguments, if they exist; else the default value for the arguments, if given in the form (arg default), else the default value for the slot, as assigned by the **defstruct** (*unknown* for the component sex in the example above), else *nil*.
- **&aux** arguments are given the default value, when in the form (arg default); when they are alone, the value of the arguments are *unassigned* (unbound). This differs from the case of **defun** where symbolic arguments to **&aux** are initialized to *nil*.

Notice that the arguments to the **:constructor** function are NOT named arbitrarily, but according to the slots to be filled. This is a requirement; using an argument name which is not a slot name will cause an error ("*unknown slot to constructor*").

(**make-name** ['s\_slot1 'g\_val1...])

- RETURNS a structure of the form of **name**.
- WHERE the values for each slot is either given using a keyword argument or defined by a default value with **defstruct**.
- EXAMPLE: see Example 2.20.

#### 14.3.1.1.3 Alterant macros

In most cases, updating slots in a structure can be done, adequately, with the **setf** function, but this can be cumbersome to use if we want to update many slots instead of just one. The keyword-driven *alterant macro* provides a tool for doing such a update.

(**alter-name** 's\_inst ['s\_slot1 'g\_val1...])

- RETURNS the structure **s\_inst** with the given slots altered.
- WHERE **s\_slot** is the slot as described in the **defstruct** not the slot as referenced by the accessor function. Each **g\_val** is evaluated **before** any slot is changed, therefore,

---

<sup>12</sup>This first occurrence of **:constructor** tells **defstruct** to create a *keyword-defined* constructor function for **person** (which will be called **make-person**). Normally this would automatically be done by **defstruct** but by using the **:constructor** argument to create additional constructor functions we turn this feature off and force **defstruct** to look for explicitly declared constructors. If we had neglected to include this, the constructor **make-person** would not have been defined.



For example, in person the slot is name and the accessor function **person-name**.

---

constructs such as the following are possible:

---

```
=> (defstruct (ship :hunk :conc-name)
      x-position
      y-position
      x-velocity
      y-velocity
      mass)
ship
=> (setq QE2 (make-ship x-position 33 y-position 12
                       x-velocity 15 y-velocity 0 mass 500000))
{33 12 15 0 500000}

; now, to exchange the values for x-position and y-position

=> (alter-ship QE2 x-position (ship-y-position QE2)
      y-position (ship-x-position QE2))
{12 33 15 0 500000}
=>
```

---

#### 14.3.1.1.4 Other caveats

Consider, for a moment, the following:

```
(defstruct box height length width).
```

This is an error. Recall that when a structure is created, accessor functions are defined to access the slots of the structure. In the case of the example, above, an accessor function, **length** was defined. It so happens that the function **length** already exists in FRANZ LISP and is used in some fairly significant ways. It is for this reason that the **:conc-name** option exists, since this provides a means for defining accessor functions which are specific for the structure defined.

## 15 The LISP stepper and fixit

- 15.1 Simple use of stepping 15-1
- 15.2 Advanced features 15-2
  - 15.2.1 Selectively turning on stepping 15-2
  - 15.2.2 Stepping with breakpoints 15-3
- 15.3 Overhead of stepping 15-3
- 15.4 Evalhook and funcallhook 15-3
- 15.5 The fixit debugger 15-4
  - 15.5.1 Interaction with trace 15-7
  - 15.5.2 Interaction with step 15-7
  - 15.5.3 Multiple error levels 15-8





## 15 The LISP stepper and fixit

Several handy debugging tools are described in detail in this chapter.

---

Note: this debugger (**step** and **debug**) has functionally been replaced by features in the current top level, **tpl**, which is described in chapter 13 (see the **tpl** commands **?debug** and **?step**).

---

### 15.1 Simple use of stepping

(**step** [t | nil | functor1 [functor2 ...]])

■ NOTE: The LISP *stepping* package is intended to give the LISP programmer a facility analogous to the Instruction Step mode of running a machine language program. The user interface is through the function (*fexpr*) **step**, which sets switches to put the LISP interpreter in and out of *stepping* mode. The most common **step** invocations follow. These invocations are usually typed at the top-level, and will take effect immediately (i.e. the next *S-expression* typed in will be evaluated in stepping mode). The facilities of this package are similar to those in the **tpl** system, but can be used separately. The capabilities of the two systems will be unified and expanded in the future.

With an argument of *t*, **step** turns stepping mode on. With an argument of *nil*, stepping mode is turned off. If any other argument(s) are provided, stepping will be turned on when an *s-expression* is read that contains that symbol in the functor position, i.e. as the **car** of a list.

■ SIDE EFFECT: In stepping mode, the LISP evaluator will print out each *S-expression* to be evaluated before evaluation, and the returned value after evaluation, calling itself recursively to display the stepped evaluation of each argument, if the *S-expression* is a function call. In stepping mode, the evaluator will wait after displaying each *S-expression* before evaluation for a command character from the console.

#### RETURN

■ Continues stepping recursively.

#### c

■ Show returned value from this level only, and continue stepping upward.



e

- Only step interpreted code.

g

- Turn off stepping mode, but continue evaluation without stepping.

n number

- Step through number evaluations without stopping.

p

- Redisplay current form in full (i.e. rebind *prinlevel* and *prinlength* to *nil*).

b

- Get breakpoint.

q

- Quit.

d

- Call debug.

## 15.2 Advanced features

### 15.2.1 Selectively turning on stepping

If

(step foo1 foo2 ...)

is typed at top level, stepping will not commence immediately, but rather when the evaluator first encounters an *S-expression* whose **car** is one of *foo1*, *foo2*, etc. This form will then display at the console, and the evaluator will be in stepping mode waiting for a command character.

Normally the stepper intercepts calls to **funcall** and **eval**. When **funcall** is intercepted, the arguments to the function have already been evaluated but when **eval** is intercepted, the arguments have not been evaluated. To differentiate the two cases, when printing the form in evaluation, the stepper prints intercepted calls to **funcall** with "*f*". Calls to **funcall** are normally caused by compiled LISP code calling other functions, whereas calls to **eval** usually occur when LISP code is interpreted. To step through only calls to **eval**, use:

(step e).



### 15.2.2 Stepping with breakpoints

Step is turned off for the duration of error breaks, but not by explicit use of the break function. Executing (step nil) inside a error loop will turn off stepping globally, i.e. within the error loop, and after return the return from the break loop.

### 15.3 Overhead of stepping

If stepping mode has been turned off by (step nil), there is no execution overhead for having the stepping packing in your LISP. If one stops stepping by typing **g**, every call to **eval** incurs a small overhead—several machine instructions, corresponding to the compiled code for a simple **cond** and one function pushdown. Running with (step foo1 foo2 ...) can be more expensive, since a **member** computation of the **car** of the current form into the list (foo1 foo2 ...) is required at each call to **eval**.

### 15.4 Evalhook and funcallhook

For **step** and potentially other user-written functions to gain control of the evaluation process, hooks were installed in the FRANZ LISP interpreter. In fact there are two hooks and they have been strategically placed in the two key functions in the interpreter: **eval** (which controls execution of interpreted code) and **funcall** (which controls compiled code if (sstatus translink nil) has been executed). The hook in **eval** is compatible with MacLisp, but there is no MacLisp equivalent of the hook in **funcall**.

To arm the hooks two forms must be evaluated: (\*rset t) and (sstatus evalhook t). Once that is done, **eval** and **funcall** do a special check when they are invoked.

If **eval** is given a form to evaluate, say (foo bar), and the symbol **evalhook** is non-*nil*, say its value is **ehook**, then **eval** will *lambda-bind* the symbols **evalhook** and **funcallhook** to *nil* and will call **ehook**, passing (foo bar) as the argument. It is **ehook**'s responsibility to evaluate (foo bar) and return its value. Typically **ehook** will call the function '**evalhook**' to evaluate (foo bar). Note that **evalhook** is a symbol whose function binding is a system function described in Chapter 4, and whose value binding, if non-*nil*, is the name of a user-written function (or a *lambda* expression, or a binary object) which will gain control whenever **eval** is called. **evalhook** is also the name of the *status* tag which must be set for all of this to work.

If **funcall** is called on a function, say **foo**, and a set of already evaluated arguments, say barv and bazv, and if the symbol **funcallhook** has a non-*nil* value, say **fhook**, then **funcall** will *lambda-bind* **evalhook** and **funcallhook** to *nil* and will call **fhook** with arguments barv, bazv and **foo**. Thus **fhook** must be a *lexpr* since it may be given any number of arguments. The function to call, **foo** in this case, will be the **last** of the arguments given to **fhook**. It is **fhook**'s responsibility to do the function call and return the value. Typically **fhook** will call the function **funcallhook** to do the **funcall**. This is an example of a **funcallhook** function which just prints the arguments on each entry to **funcall** and the return value.



---

```
=> (defun fhook n
      (let ((form (cons (arg n) (listify (1- n))))
            (retval))
        (patom "calling ") (print form) (terpr)
        (setq retval (funcallhook form 'fhook))
        (patom "returns ") (print retval) (terpr)
        retval))

fhook
=> (*rset t)
=> (sstatus evalhook t)
=> (sstatus translink nil)
=> (setq funcallhook 'fhook)
calling (print fhook) ;; now all compiled code is traced
fhookreturns nil
calling (terpr)

returns nil
calling (patom "--> ")
=> returns "--> "
calling (read nil Q00Q00)
(array foo t 10) ;; to test it, we see what happens when
returns (array foo t 10) ;; we make an array
calling (eval (array foo t 10))
calling (append (10) nil)
returns (10)
calling (lessp 1 1)
returns nil
calling (apply times (10))
returns 10
calling (small-segment value 10)
calling (boole 4 137 127)
returns 128

... there is plenty more ...
```

---

## 15.5 The fixit debugger

Fixit is a debugging environment for FRANZ LISP written and documented by David S. Touretzky of Carnegie-Mellon University for MacLISP, and adapted to FRANZ LISP by Mitch Marcus of Bell Laboratories. One of Fixit's goals is to get a program being tested running again as quickly as possible. The user is assisted in making changes to his functions "on the fly," i.e. in the midst of execution, and then computation is resumed.

To enter the debugger type (debug). The debugger goes into its own *read-eval-print* loop. Like the top-level, the debugger understands certain special commands. One of these is help, which prints a list of the available commands. The basic idea is that you are somewhere in a stack of calls to eval. The command **bka** is probably the most appropriate for looking at the stack. There are commands to move up and down. If you



want to know the value of *x* as of some place in the stack, move to that place and type *x* or *(cdr x)* or anything else that you might want to evaluate. All evaluation is done as of the current stack position. You can fix the problem by changing the values of variables, editing functions or expressions in the stack etc. Then you can continue from the current stack position (or anywhere else) with the **redo** command. Or you can simply return the right answer with the **return** command.

When it is not immediately obvious why an error has occurred or how the program got itself into its current state, **Fixit** comes to the rescue by providing a powerful debugging loop in which the user can:

- examine the stack;
- evaluate expressions in context;
- enter stepping mode;
- restart the computation at any point.

The result is that program errors can be located and fixed more rapidly.

The debugger can only work effectively when extra information is kept about forms in evaluation by the LISP system. Evaluating *(\*rset t)* tells the LISP system to maintain this information. If you are debugging compiled code you should also be sure that the execute *(sstatus translink nil)*.

**(debug [s\_msg])**

■ **NOTE:** Within a program, you may enter a debug loop directly by putting in a call to **debug** where you would normally put a call to **break**. Also, within a break loop you may enter **Fixit** by typing **debug**. If an argument is given to **debug**, it is treated as a message to be printed before the debug loop is entered. Thus you can put *(debug [just before loop])* into a program to indicate what part of the program is being debugged.

**TOP**

- Go to top of stack (latest expression).

**BOT**

- Go to bottom of stack (first expression).

**P**

- Show current expression (with ellipsis).

**PP**

- Show current expression in full.



**WHERE**

- Give current stack position.

**HELP**

H  
?

- Types the abbreviated command summary.

**U [n] [f]**

- Go up n (default 1) stack frames. If f is given, go up n (default 1) occurrences of the function f.

**UP n**

- Go up n (default 1) user-written functions.

**DN n**

**DNFN n**

- Go down n (default 1) user-written functions.

**OK**

- Resume processing; continue after an error or debug loop.

**REDO**

- Restart the computation with the current stack frame. The **OK** command is equivalent to **TOP** followed by **REDO**.

**REDO f**

- Restart the computation with the last call to function f; the stack is searched downward from the current position.

**STEP**

- Restart the computation at the current stack frame, but first turn on stepping mode (assumes the stepper is loaded).

**RETURN e**

- Return from the current position in the computation with the value of expression e.

**BKsuffix [n] [f]**

■ This is the general backtrace command. The optional **suffix** is a combination of one or more of the letters **A**, **F**, **V**, **E**, and **C**. The meaning of the suffixes is explained below. With no suffix, only user-defined functions are shown. If **n** is given, **n** levels of the stack are shown. If **f** is given, only calls to function **f** are shown.

**BKF**

■ Print a backtrace showing function names instead of expressions.

**BKA**

■ Print a backtrace showing all functions and expressions, not just user-written ones.

**BKV**

■ Print a backtrace showing variable bindings as well as functions and expressions.

**BKE**

■ Print a backtrace showing everything in the expression, i.e. without ellipses.

**BKC**

■ Print a backtrace going no further than the current position on the stack.

### 15.5.1 Interaction with trace

**Fixit** knows about the standard FRANZ LISP trace package, and tries to make tracing invisible while in the debug loop. However, because of the way **trace** works, it may sometimes be the case that the functions on the stack are really uninterned atoms that have the same name as a traced function. (This only happens when a function is traced wherein another one.) **Fixit** will call attention to **trace**'s hackery by printing an appropriate tag next to these stack entries.

### 15.5.2 Interaction with step

The **step** function may be invoked from within **Fixit** via the **STEP** command. **Fixit** initially turns off stepping when the debug loop is entered. If you step through a function and get an error, **Fixit** will still be invoked normally. At any time during stepping, you may explicitly enter **Fixit** via the **D** (debug) command.



### 15.5.3 Multiple error levels

**Fixit** will evaluate arbitrary LISP expressions in its debug loop. The evaluation is not done within an **errset**, so, if an error occurs, another invocation of the debugger can be made. When there are multiple errors on the stack, **Fixit** displays a barrier symbol between each level that looks something like

<-----UDF-->

The *UDF* in this case stands for UnDefined Function. Thus, the upper level debug loop was invoked by an undefined function error that occurred while in the lower loop.

## 16 The LISP editor

- 16.1 Introduction 16-1
- 16.2 Tutorial 16-1
- 16.3 Editor functions 16-5
  - 16.3.1 The edit chain 16-6
    - 16.3.1.1 Edit chain command summary 16-7
- 16.4 Printing commands 16-7
- 16.5 Scope of attention 16-8
- 16.6 Pattern and search commands 16-10
  - 16.6.1 Pattern specification 16-10
  - 16.6.2 Search commands 16-11
- 16.7 Location specifications 16-12
- 16.8 Structure modification commands 16-13
  - 16.8.1 Structure modification commands 16-13
  - 16.8.2 Extraction and embedding commands 16-15
  - 16.8.3 Move and copy commands 16-15
- 16.9 List modification commands 16-16
  - 16.9.1 Parentheses-moving commands 16-16
  - 16.9.2 To-and-thru commands 16-17
- 16.10 Undoing commands 16-18
- 16.11 Commands that evaluate 16-19
- 16.12 Commands that test 16-20
- 16.13 Editor macros 16-20
- 16.14 Miscellaneous editor commands 16-22





## 16 The LISP editor

### 16.1 Introduction

Many people use standard text editors to edit their LISP programs. However there are also LISP *structure-oriented* embedded editors which are particularly handy for the editing of LISP programs and data. These operate in a rather different fashion, namely within a LISP environment. Such an editor is handy for rapid fixes and reevaluating of tests without exiting from the LISP system. For example, you can fix a bug and then continue your computation from a break-point. The editor has its own command structure which includes the ability to evaluate arbitrary LISP expressions.

The LISP editor **editf** and its related components in FRANZ LISP differ from file/text editors in that editor commands directly change the internal structure of LISP expressions rather than an external character representation. In particular, it is not possible for the LISP editor to create an expression with unbalanced parentheses because such expressions cannot occur in the internal representation of a LISP object. This editor modifies the structure of existing LISP objects but does not automatically update any copies of the objects on files. See, for example, the function **pp** in chapter 5, for writing functions to files.

This editor is based on the InterLISP editor and has an almost identical command syntax.

### 16.2 Tutorial

Suppose that we wish to define a function **foo** which adds five to its argument if it is a number, and returns *nil* otherwise. We might type the following (incorrect) expression into the interpreter:

---

```
=> (defun foo (x) ; incorrect
      ((numberp x) (plus x 5))
      (t nil))
foo
```

---

Executing **foo** will cause an error because the conditional function **cond** has been left out. We can correct it by editing the function **foo**:



---

```
=> (editf foo)
edit
#
```

---

We are now in edit mode, with the attention of the editor focused on the expression which defines **foo**. To print the expression on the screen, type:

---

```
#p
(lambda (x) (& & ) (t nil))
```

---

This is not exactly what was typed in. **defun** is really a macro which expands into something involving **def** and **lambda**, so that is why the **lambda** is there. The comment has been omitted and the spacing is different. The reason for these differences is that we are editing a LISP object, and not the characters which were typed to define the LISP object. The symbol **&** is just a shorthand for a more complicated subexpression. To see the full expression, type:

---

```
#?
(lambda (x) ((numberp x) (plus x 5)) (t nil))
```

---

This is the current expression being edited. To insert **cond** before the third element in the current expression, type:

---

```
#(-3 cond)
(lambda (x) cond (& & ) (t nil))
```

---

Now we need a pair of parentheses. The editor requires that they be entered as a pair. To insert a left parenthesis before the third element of the current expression and a matching right parenthesis at the end, type:

---

```
#{li 3)
(lambda (x) (cond & & & & ))
```

---

The expression appears even more abbreviated as the default print function only shows parenthetical nesting up to a level of two. For the full expression, type:

---

```
#?  
(lambda (x) (cond ((numberp x) (plus x 5)) (t nil)))
```

---

This definition for **foo** will work, so we can save the change and return to LISP by typing:

---

```
#ok  
foo  
=> (foo 20)  
25  
=> (foo 'not-a-number)  
nil  
=>
```

---

Now suppose that we wish to change **foo** so that it adds ten instead of adding five. We reënter the editor:

---

```
=> (editf foo)  
edit  
#?  
(lambda (x) (cond ((numberp x) (plus x 5)) (t nil)))
```

---

The current expression only has three elements and 5 is not one of them, so we cannot change 5 directly. Typing 3 causes the editor to focus attention on the third element, and to consider that to be the current expression.

---

```
#3  
(cond ((numberp x) (plus x 5)) (t nil))  
#2  
((numberp x) (plus x 5))  
#2  
(plus x 5)
```

---

The following command replaces the third element with a 10.



---

```
#(3 10)
(plus x 10)
```

---

Typing 0 (zero) takes us to a higher level:

---

```
#0
((numberp x) (plus x 10))
#0
(cond ((numberp x) (plus x 10)) (t nil))
#0
(lambda (x) (cond ((numberp x) (plus x 10)) (t nil)))
```

---

Suppose that we wish to change **foo** so that it returns "*not-a-number*" if the argument is not a number. A quick way to find *nil* in the current expression is to type:

---

```
#f nil
```

---

The current expression is a valid LISP object, but it is called the *tail* of an expression because a left parenthesis would be misleading. The following commands replace *nil*, check the result, and exit the editor.

---

```
#(1 'not-a-number)
#^
(lambda (x) (cond & & & & ))
#?
(lambda (x) (cond ((numberp x) (plus x 10)) (t 'not-a-number)))
#ok
=>
```

---

Variable values and property lists can also be edited. The following example illustrates assigning a value and a property list to a variable, and then using the editor to make modifications.

---

```
=> (setq foo '(this is a chair))
(this is a chair)
=> (putprop 'foo 'blue 'color)
color
=> foo
```

---

```

(this is a chair)
=> (get 'foo 'color)
blue
=> (editv foo)
edit
#p
(this is a chair)
#(4 pillow)
pillow
#p
(this is a pillow)
#ok
foo
=> (editp foo)
edit
#p
(color blue)
#(2 red)
(color red)
#ok
foo
=> (get 'foo 'color)
red

```

---

While within the editor, you can reverse the most recent change, type the command **undo**. The command **lundo** undoes all changes made during the editing session.

### 16.3 Editor functions

**(editf s\_x1 ...)**

- **SIDE EFFECT:** Edits a function with the name **s\_x1**. Any additional arguments are optional commands to the editor.
- **RETURNS** **s\_x1**.
- **NOTE:** If **s\_x1** is not an editable function, **editf** generates a "fn not editable" error.

**(editv s\_var [ g\_com1 ... ])**

- **SIDE EFFECT:** Edits values in a manner similar to the way **editf** edits functions. The value of the variable can be changed by subsequent editing commands.
- **RETURNS** the name of the variable whose value was edited.

**(editp s\_x)**

- **SIDE EFFECT:** Edits property lists.
- **RETURNS** the atom whose property list was edited.



(editfns s\_x [ g\_coms1 ... ])

■ **SIDE EFFECT:** Performs the same editing operations on several functions. The symbol **s\_x** is the function or list of functions, and the following arguments are the editing commands. Evaluation of **editfns** will map down the list of functions, print the name of each function, and call the editor (via **editf**) on each function.

■ **RETURNS** *nil*.

■ **EXAMPLE:** (editfns foofns (r fie fum)) will change every *fie* to *fum* in each of the functions in the list called *foofns*.

■ **NOTE:** The call to the editor is **errset** protected, so that if the editing of one function causes an error, **editfns** will proceed to the next function. In the above example, if one of the functions did not contain a *fie*, the *r* command would cause an error, but editing would continue with the next function.

(editracefn s\_com)

■ **NOTE:** This is available to help the user debug complex edit macros, or subroutine calls to the editor. It is initially an undefined function, to be defined by the user. Whenever the value of **editracefn** is non-*nil*, the editor calls the function **editracefn** before executing each command (at any level), giving it that command as its argument.

(editfindp x pat nil)

■ **NOTE:** Allows a program to use the editor find command as a pure predicate from outside the editor. It searches for the pattern *pat* in the expression *x*.

■ **RETURNS** *t* if the editor command **f pat** would succeed, *nil* otherwise.

### 16.3.1 The edit chain

The edit-chain is a list whose first element is the expression you are now editing (*current expression*), the next element is what would become the current expression if you were to type a **0**, etc., until the last element which is the expression that was passed to the editor.

(\# g\_com1 ...)

■ **RETURNS** what the current expression would be after executing the edit commands *g\_com1 ...* starting from the present edit chain, generating an error if any of *g\_comi* cause errors. The current edit chain is never changed.

■ **EXAMPLE:** (i r (quote x) (\# (cons ..z))) replaces all *xs* in the current expression by the first **cons** containing a *z*.



### 16.3.1.1 Edit chain command summary

**mark**

- Adds the current edit chain to the front of the list marklst.

**\_**

- Makes the new edit chain be (car marklst).

**(\_ pattern)**

- Ascends the edit chain looking for a link which matches pattern.

**--**

- A double underscore is similar to a single underscore **\_** but also erases the mark.

**/**

- Makes the edit chain be the value of **unfind**. **unfind** is set to the current edit chain by each command that makes a "big jump," i.e. a command that usually performs more than a single ascent or descent, namely **^**, **\_**, **\_\_**, **!nx**, all commands that involve a search (e.g., **f**, **lc**, **!!**, **below**), and **/** and **/p** themselves. If the user types **f cond**, and then **f car**, **/** would take him back to the **cond**. Another **/** would take him back to the **car**, etc.

**/p**

- Restores the edit chain to its state as of the last print operation. If the edit chain has not changed since the last printing, **/p** restores it to its state as of the printing before that one. If the user types **p** followed by **3 2 1 p**, **/p** will return to the first **p**, i.e. would be equivalent to **0 0 0**. Another **/p** would then take him back to the second **p**.

## 16.4 Printing commands

**p**

- Prints current expression in abbreviated form.

**(p m)**

- Prints **m**th element of current expression in abbreviated form.



(p m n)

- Prints mth element of current expression as though *printlev* were given a depth of n. (p 0 n) prints the current expression as though *printlev* were given a depth of n.

(p foo)

- Searches for the first occurrence of foo and then prints it.

?

- Prints the current expression as though *printlev* were given a depth of 100.

pp

- Pretty-prints the current expression.

pp\*

- is like pp, but forces comments to be shown.

## 16.5 Scope of attention

Attention-changing commands allow you to look at a different part of a LISP expression you are editing. The substructure upon which the editor's attention is centered is called *the current expression*. Changing the current expression means shifting attention and not actually modifying any structure.

n

- WHERE n is greater than zero.
- Makes the nth element of the current expression be the new current expression.

-n

- WHERE n is greater than zero.
- Makes the nth element from the end of the current expression be the new current expression.

0

- Makes the next higher expression be the new correct expression. If the intention is to go back to the next higher left parenthesis, use the command !0.

up

- Unless the current expression is a tail, **up** changes the current expression to the one which has the previous current expression as its first element. Tails are unchanged. (A tail is an expression which starts with "..." when printed with the p

command.)

**!0**

- Goes back to the next higher left parenthesis.

- Makes the top level expression be the current expression.

**nx**

- Makes the current expression be the next expression. It will not go through an unmatched right parenthesis, so it generates an error if the current expression is the last.

**(nx n)**

- WHERE  $n$  is greater than zero.
- EQUIVALENT TO:  $n$  consecutive **nx** commands.

**lnx**

- Makes current expression be the next expression at a higher level. Goes through any number of right parentheses to get to the next expression. It always gives a different result from **nx**.

**bk**

- Makes the current expression be the previous expression in the next higher expression.

**(nth n)**

- WHERE  $n$  is greater than zero.
- Makes the list starting with the  $n$ th element of the current expression be the current expression.

**(nth \$)**

- This generalized **nth** command locates \$, and then backs up to the current level, where the new current expression is the tail whose first element contains, however deeply, the expression that was the terminus of the location operation.

**!!**

- As in (pattern !! . \$), searches for an expression or tail which starts with pattern and ends with \$. For example, (cond !! return) finds a **cond** that contains a return, at any depth.



(below com x)

■ This ascends to higher levels searching for com and then changes the current expression to the one which is x levels below com. The default value of x is 1. For example (below cond) will cause the **cond** clause containing the current expression to become the new current expression.

(nex x)

■ EQUIVALENT TO: (below x) followed by **nx**. For example, if you are deep inside of a **selectq** clause, you can advance to the next clause with (nex selectq).

**nex**

■ The atomic form of **nex** is useful if you will be performing repeated executions of (nex x). By simply marking the chain corresponding to x, you can use **nex** to step through the sublists.

## 16.6 Pattern and search commands

In many of the editor commands it is possible to specify a pattern to direct an operation to a subexpression or change the attention of the editor. This section describes the types of patterns and searches.

### 16.6.1 Pattern specification

A pattern pat matches with x if:

- pat is **eq** to x. In this case, x may not be a tail, so (a b) will not match ... a b).
- x is a list, (car pat) matches (car x), and (cdr pat) matches (cdr x).
- pat is **&**.
- pat is a number and **equal** to x.
- (car pat) is the atom *\*any\**, (cdr pat) is a list of patterns, and one of those patterns matches x.
- pat is a literal atom or string, and (nthchar pat -1) is **@**, then pat matches with any literal atom or string which has the same initial characters as pat, e.g. *ver@* matches with *verylongatom*, as well as *"verylongstring"*.
- if (car pat) is the atom **--**, pat matches x if: (cdr pat) is *nil*, i.e. pat is **--** (e.g. (a --) matches (a), (a b c), and (a . b)), in other words -- can match any tail of a list; or (cdr pat) matches with some tail of x (e.g. (a -- (&)) will match with (a b c (d)), but not (a b c d), or (a b c (d) e)). However, note that (a -- (& --)) will match with (a b c (d) e). In other words, -- will match any interior segment of a list.
- if (car pat) is the atom **==**, pat matches x if and only if (cdr pat) is **eq** to x. (This pattern is for use by programs that call the editor as a subroutine, since any non-atomic expression in a command typed in by the user obviously cannot be **eq** to existing structure.)



- **pat** has **!!!** for its **car**, and either its **cdr** matches with **x** or **x** is a tail which would match if it had a left parenthesis. For example, searching for a match with **(!!! b c)** will succeed on **(a (b c))** as well as on **(a b c)**.

## 16.6.2 Search commands

### **f pattern**

- Finds the next instance of **pattern**. If no **pattern** is given then the last **pattern** is used.

### **(f pattern n)**

- Finds the next instance of **pattern**. (Here, **n** stands for next, and not an integer.)

### **(f pattern t)**

- Similar to **f pattern**, except, for example, if the current expression is **(cond ..)**, **f cond** will look for the next **cond**, but **(f cond t)** will not.

### **(f pattern n)**

- WHERE **n** is greater than zero.
- Finds the **n**th place that **pattern** matches. If the current expression is **(foo1 foo2 foo3)**, **(f foo@ 3)** will find **foo3**.

### **(f pattern)**

### **(f pattern nil)**

- Only matches with elements at the top level of the current expression. If the current expression is **(prog nil (setq x (cond & &)) (cond &) ...)** **f (cond --)** will find the **cond** inside the **setq**, whereas **(f (cond --))** will find the top-level **cond**, i.e. the second one.

### **(fs pattern1 ... patternn)**

- Is equivalent to **f pattern1** followed by **f pattern2** . . . followed by **f patternn**, so that if a search fails, the edit chain is left at the place where the previous **pattern** matched.

### **(f= expression x)**

- Searches for a structure **eq** to **expression**.



(orf pattern1 ... patternn)

- Searches for an expression that is matched by one of the pattern*i*.

bf pattern

- This backwards find searches for the first previous occurrence of the pattern. If the current expression is the top-level expression, then the entire expression is searched in reverse print order. For example, if the current expression is (prog nil (setq x (setq y (list z))) (print x)), then f list followed by bfsetq will change the current expression to (setq y (list z)), as will f print followed by bfsetq.

(bf pattern t)

- This is similar to the above backwards find. Search always includes current expression, i.e., starts at end of current expression and works backward, then ascends and backs up, etc.

## 16.7 Location specifications

Many editor commands use a method of specifying position called a location specification. The meta-symbol \$ is used to denote a location specification. \$ is a list of commands interpreted as described above. \$ can also be atomic, in which case it is interpreted as (list \$). A location specification is a list of edit commands that are executed in the normal fashion with the following exception. All commands not recognized by the editor are interpreted as though they had been preceded by f. The location specification (cond 2 3) specifies the third element in the first clause of the next **cond**.

The if command and the \# function provide a way of using in location specifications arbitrary predicates applied to elements in the current expression.

\$

- In descriptions of the editor, the meta-symbol \$ is used to denote a location specification. \$ is a list of commands interpreted as described above. \$ can also be atomic.

(lc . \$)

- Provides a way of explicitly invoking the location operation. (lc cond 2 3) will perform a search for a **cond** clause and then change the current expression to the third element of the **cond** clause.

(lcl . \$)

- Same as lc except search is confined to current expression. To find a **cond** containing a **return**, one might use the location specification (cond (lcl return) /) where the / would reverse the effects of the lcl command, and make the final current expression be the **cond**.



**(second . \$)**

- Same as (lc . \$) followed by another (lc . \$) except that if the first succeeds and second fails, no change is made to the edit chain.

**(third . \$)**

- Similar to **second**.

**16.8 Structure modification commands**

All structure modification commands are undoable. See section 16.11 for a description of undoing commands.

In **insert**, **delete**, **replace** and **change**, if \$ is *nil* (empty), the corresponding operation is performed on the current edit chain, i.e. (replace with (car x)) is equivalent to (! (car x)). For added readability, here is also permitted, e.g. (insert (print x) before here) will insert (print x) before the current expression (but not change the edit chain). It is perfectly legal to ascend to insert, replace, or delete. For example (insert (return) after ^ prog -1) will go to the top, find the first **prog**, and insert a (return) at its end, and not change the current edit chain.

The **a**, **b**, and **!** commands all make special checks in e1 thru em for expressions of the form (\# . coms). In this case, the expression used for inserting or replacing is a copy of the current expression after executing coms, a list of edit commands. (insert (\# f cond -1 -1) after3) will make a copy of the last form in the last clause of the next **cond**, and insert it after the third element of the current expression.

**16.8.1 Structure modification commands****(n)**

- WHERE **n** is greater than one.
- Deletes the corresponding element from the current expression.

**(n e1 ... em)**

- WHERE **n** and **m** are greater than one.
- Replaces the **n**th element in the current expression with e1 ... em.

**(-n e1 ... em)**

- WHERE **n** and **m** are greater than one.
- Inserts e1 ... em before the **n**th element in the current expression.



(n e1 ... em)

- WHERE the letter **n** stands for *next* or *nconc*, not a number. **m** is greater than one.
- Attaches e1 ... em at the end of the current expression.

(a e1 ... em)

- Inserts e1 ... em after the current expression (or after its first element if it is a tail).

(b e1 ... em)

- Inserts e1 ... em before the current expression. To insert foo before the last element in the current expression, perform -1 and then (b foo).

(! e1 ... em)

- Replaces the current expression by e1 ... em. If the current expression is a tail then replace its first element.

(r x y)

- Replaces each occurrence of **x** with **y** in the current expression. The term **x** can be an atom, a list, or a location specification.

(sw n m)

- Switches the **n**th and **m**th elements of the current expression. For example, if the current expression is (list (cons (car x) (car y)) (cons (cdr x) (cdr y))), (sw 2 3) will modify it to be (list (cons (cdr x) (cdr y)) (cons (car x) (car y))). (sw car cdr) would produce the same result.

**delete**

(!)

- Deletes the current expression, or if the current expression is a tail, deletes its first element.

(delete . \$)

- Does a (lc . \$) followed by delete. The current edit chain is not changed.

(insert e1 ... em before . \$)

- Similar to (lc . \$) followed by (b e1 ... em).

(insert e1 ... em after . \$)

- Similar to **insert before** except uses **a** instead of **b**.

(insert e1 ... em for . \$)

- Similar to **insert before** except uses ! for b.

(replace \$ with e1 ... em)

- Here \$ is the segment of the command between **replace** and **with**.

(change \$ to e1 ... em)

- Same as **replace with**.

## 16.8.2 Extraction and embedding commands

(xtr . \$)

- Replaces the original current expression with the expression that is current after performing (lcl . \$).

(mbd x)

- If x is a list, substitutes the current expression for all instances of the atom \* in x, and replaces the current expression with the result of that substitution. If x is atomic, (mbd x) is the same as (mbd (x \*)).

(extract \$1 from \$2)

- This is an editor command which replaces the current expression with one of its subexpressions (from any depth). \$1 is the segment between **extract** and **from**. For example, if the current expression is (print (cond ((null x) y) (t z))) then following (extract y from cond), the current expression will be (print y). (extract 2 -1 from cond), (extract y from 2), (extract 2 -1 from 2) will all produce the same result.

(embed \$ in . x)

- Replaces the current expression with a new expression which contains it as a subexpression. \$ is the segment between **embed** and **in**.
- EXAMPLE: (embed print in setq x), (embed 3 2 in return), (embed cond 3 1 in (or \* (null x))).

## 16.8.3 Move and copy commands

(move \$1 to com . \$2)

- \$1 is the segment between **move** and **to**. Where com is before, after, or the name of a list command, e.g., :, n, etc. If \$2 is nil, or (here), the current position



specifies where the operation is to take place. If \$1 is *nil*, the **move** command allows the user to specify some place the current expression is to be moved to. If the current expression is (a b d c), (move 2 to after 4) will make the new current expression be (a c d b).

(mv com . \$)

- EQUIVALENT TO: (move here to com . \$).

(copy \$1 to com . \$2)

- Like **move** except that the source expression is not deleted.

(cp com . \$)

- Like **mv** except that the source expression is not deleted.

## 16.9 List modification commands

The commands presented in this section permit modification of the list structure itself, as opposed to modifying components. Their effect can be described as inserting or removing a single left or right parenthesis, or pair of left and right parentheses. Some people find that use of only **bi** and **bo** to be less confusing and quite adequate for use instead of the 4 additional commands.

### 16.9.1 Parentheses-moving commands

(bi n m)

- This “both in” command inserts parentheses before the nth element and after the mth element in the current expression.
- EXAMPLE: If the current expression is (a b (c d e) f g), then (bi 2 4) will modify it to be (a (b (c d e) f) g). (bi n) is the same as (bi n n). If the current expression is (a b (c d e) f g), then (bi -2) will modify it to be (a b (c d e) (f) g).

(bo n)

- This “both out” command removes both parentheses from the nth element.
- EXAMPLE: If the current expression is (a b (c d e) f g), then (bo d) will modify it to be (a b c d e f g).

(li n)

- This “left in” command inserts a left parenthesis before the nth element and a matching right parenthesis at the end of the current expression.



■ **EXAMPLE:** If the current expression is (a b (c d e) f g), then (li 2) will modify it to be (a (b (c d e) f g)).

(lo n)

■ This "left out" command removes a left parenthesis from the nth element. all elements following the nth element are deleted.

■ **EXAMPLE:** If the current expression is (a b (c d e) f g), then (lo 3) will modify it to be (a b c d e).

(ri n m)

■ This "right in" command moves the right parenthesis at the end of the nth element in to after the mth element. Inserts a right parenthesis after the mth element of the nth element. The rest of the nth element is brought up to the level of the current expression.

■ **EXAMPLE:** If the current expression is (a (b c d e) f g), (ri 2 2) will modify it to be (a (b c) d e f g).

(ro n)

■ This "right out" command moves the right parenthesis at the end of the nth element out to the end of the current expression. Removes the right parenthesis from the nth element, moving it to the end of the current expression. All elements following the nth element are moved inside of the nth element.

■ **EXAMPLE:** If the current expression is (a b (c d e) f g), (ro 3) will modify it to be (a b (c d e f g)).

### 16.9.2 To-and-thru commands

Certain commands can be made to operate on several contiguous elements of a list by using the to or thru command in their respective location specifications. These commands are **to**, **thru**, **extract**, **embed**, **delete**, **replace**, and **move**. **to** and **thru** can also be used directly with **xtr** (which takes after a location specification), as in (xtr (2 thru 4)) (from the current expression).

(\$1 to \$2)

■ Same as **thru** except last element not included.

(\$1 to)

■ **EQUIVALENT TO:** (\$1 thru -1).



(\$1 thru \$2)

■ If the current expression is (a (b (c d) (e) (f g h) i) j k), following (c thru g) the current expression will be ((c d) (e) (f g h)). If both \$1 and \$2 are numbers, and \$2 is greater than \$1, then \$2 counts from the beginning of the current expression, the same as \$1. In other words, if the current expression is (a b c d e f g), (3 thru 4) means (c thru d), not (c thru f). In this case, the corresponding bi command is (bi 1 \$2-\$1+1).

(\$1 thru)

■ EQUIVALENT TO: (\$1 thru -1).

## 16.10 Undoing commands

Each command that causes structure modification automatically adds an entry to the front of a list called undolst. The **undo** command undoes the most recent such command based on information in undolst.

**undo**

■ The **undo** command undoes most recent structure modification command that has not yet been undone, and prints the name of that command, e.g. "*mbd undone*". The edit chain is then exactly what it was before the 'undone' command had been performed.

**!undo**

■ Undoes all modifications performed during this editing session, i.e. this call to the editor.

**unblock**

■ Removes an *undo-block*. If executed at a non-blocked state, i.e. if **undo** or **!undo** could operate, types not blocked.

**test**

■ Adds an undo-block at the front of undolst. Note that By using test together with **!undo**, the user can perform a number of changes, and then undo all of them with a single **!undo** command.

**??**

■ Prints the entries on undolst. The entries are listed most recent entry first.



## 16.11 Commands that evaluate

These commands allow you to execute arbitrary LISP expressions, perhaps including calling a function you are editing! All the changes you have made are *in place* in the interpreted version of the function under edit.

**e**

- When typed in as a single atomic command, passes the next *s-expression* to the LISP reader and evaluates and prints it. Other uses of the symbol **e** are unaffected, e.g. (insert d before e) will treat e as a pattern.

**(e x)**

- Evaluates x and prints the result.

**(e x t)**

- The same as (e x) but does not print.

**(i c x1 ... xn)**

- Same as (c y1 ... yn) where yi is (eval xi).
- **EXAMPLE:** (i 3 (cdr foo)) will replace the third element of the current expression with the **cdr** of the value of foo. (i n foo (car fie)) will attach the value of foo and **car** of the value of fie to the end of the current expression. (i f= foo t) will search for an expression **eq** to the value of foo. If c is not an atom, it is evaluated as well. (The **coms** and **comsq** commands below provide more general ways of computing commands.)

**(coms x1 ... xn)**

- Each xi is evaluated and its value executed as a command. For example, (coms (cond (x (list 1 x)))) will replace the first element of the current expression with the value of x if non-*nil*, otherwise do nothing. (Note that *nil* as a command does nothing.)

**(comsq com1 ... comn)**

- Executes com1 ... comn and used mainly useful in conjunction with the **coms** command. For example, suppose the user wishes to compute an entire list of commands for evaluation, as opposed to computing each command one at a time as does the **coms** command. He would then write (coms (cons 'comsq x)) where x computed the list of commands, e.g., (coms (cons 'comsq (get foo 'commands))).



## 16.12 Commands that test

(if x)

- Generates an error unless the value of (eval x) is non-*nil*. Thus an error is generated if either (eval x) causes an error or if (eval x) is *nil*.

(if x coms1)

- Evaluates x and if it is non-*nil*, executes coms1. Otherwise, generates an error.

(if x coms1 coms2)

- Evaluates x and if it is non-*nil*, executes coms1. If (eval x) causes an error or is equal to *nil*, coms2 is executed.

(lp . coms)

- Repeatedly executes coms, a list of commands, until an error occurs.
- EXAMPLE: (lp f print (n t)) will attach a t at the end of every print expression. (lp f print (if (\# 3) nil ((n t)))) will attach a t at the end of each print expression which does not already have a second argument. (That is the form (\# 3) will cause an error if the edit command 3 causes an error, thereby selecting ((n t)) as the list of commands to be executed. The if could also be written as (if (caddr (\#)) nil ((n t))).)

(lpq . coms)

- Same as lp but does not print n occurrences.

(orr coms1 ... comsn)

- orr begins by executing coms1, a list of commands. If no error occurs, orr is finished. otherwise, orr restores the edit chain to its original value, and continues by executing coms2, etc. If none of the command lists execute without errors, i.e. the orr "drops off the end," orr generates an error. Otherwise, the edit chain is left as of the completion of the first command list which executes without error.

## 16.13 Editor macros

Many of the more sophisticated branching commands in the editor, such as **orr**, **if**, etc., are most often used in conjunction with edit macros. The macro feature permits the user to define new commands and thereby expand the editor's repertoire. (However, built-in commands always take precedence over macros, i.e. the editor's repertoire can be expanded, but not modified.) Macros are defined by using the **m** command. If a macro is redefined, its new definition replaces its old.



**(m c . coms)**

■ Defines **c** as an atomic command, where **c** is an atom and **coms** is a list. Executing **c** is then the same as executing the list of commands **coms**. see the next paragraph for an example. Macros can also define list commands, i.e. commands that take arguments.

■ EXAMPLE: **(m (c) (arg[1] ... arg[n]) . coms)**, where **c** is an atom. **m** defines **c** as a list command. Executing **(c e1 ... en)** is then performed by substituting **e1** for **arg[1]**, ... **en** for **arg[n]** throughout **coms**, and then executing **coms**. A list command can be defined via a macro so as to take a fixed or indefinite number of 'arguments.' The form given above specified a macro with a fixed number of arguments, as indicated by its argument list. If the 'argument list' is atomic, the command takes an indefinite number of arguments.

□ **(m (c) args . coms)** where **c** and **args** are both atoms, defines **c** as a list command. Executing **(c e1 ... en)** is performed by substituting **(e1 ... en)**, i.e. **cdr** of the command, for **args** throughout **coms**, and then executing **coms**.

**(m bp bk up p)**

■ Defines **bp** as an atomic command which does three things, a **bk**, an **up**, and a **p**. Note that macros can use commands defined by macros as well as built in commands in their definitions. For example, suppose **z** is defined by **(m z -1 (if (null (\#)) nil (p)))**, i.e. **z** does a **-1**, and then if the current expression is not *nil*, a **p**. Now we can define **zz** by **(m zz -1 z)**, and **zzz** by **(m zzz -1 -1 z)** or **(m zzz -1 zz)**. We could define a more general **bp** by **(m (bp) (n) (bk n) up p)**. **(bp 3)** would perform **(bk 3)**, followed by an **up**, followed by a **p**. The command **2nd** can be defined as a macro by **(m (2nd) x (orr ((lc . x) (lc . x))))**.

□ Note that for all editor commands, 'built in' commands as well as commands defined by macros, atomic definitions and list definitions are completely independent. In other words, the existence of an atomic definition for **c** in no way affects the treatment of **c** when it appears as **car** of a list command, and the existence of a list definition for **c** in no way affects the treatment of **c** when it appears as an atom. In particular, **c** can be used as the name of either an atomic command, or a list command, or both. In the latter case, two entirely different definitions can be used. Note also that once **c** is defined as an atomic command via a macro definition, it will not be searched for when used in a location specification, unless **c** is preceded by an **f**. **(insert -- before bp)** would not search for **bp**, but instead perform a **bk**, an **up**, and a **p**, and then do the insertion. The corresponding also holds true for list commands.

**(bind . coms)**

■ This is an edit command which is useful mainly in macros. It binds three dummy variables **#1**, **#2**, **#3**, (initialized to *nil*), and then executes the edit commands **coms**. Note that these bindings are only in effect while the commands are being executed, and that **bind** can be used recursively; it will rebind **#1**, **#2**, and **#3** each time it is invoked.



usermacros

- This is a LISP variable which contains a list of the user-defined editing macros with their definitions. These macros remain in effect from one editing session to another. you can save your macros for another LISP session by saving usermacros on a disk file.

editcomsl

- This is a LISP variable which contains a list of the "list commands" recognized by the editor. (These are the commands such as **ll** whose execution takes the form (command arg1 arg2 ...).)

## 16.14 Miscellaneous editor commands

This section contains a descriptions of those editing functions which can be called from the LISP top level. These include functions for merely entering the editor as well as some which perform some editing tasks and return to the top level.

ok

- Exits from the editor.

nil

- Unless preceded by **f** or **bf**, is always a null operation.

tty

- Calls the editor recursively. The user can then type in commands, and have them executed. The **tty** command is completed when the user exits from the lower editor (with **ok** or **stop**). The **tty** command is extremely useful. It enables the user to set up a complex operation, and perform interactive attention-changing commands part way through it. For example the command (move 3 to after cond 3 p tty) allows the user to interact, in effect, within the **move** command. He can verify for himself that the correct location has been found, or complete the specification "by hand." In effect, **tty** says "I'll tell you what you should do when you get there."

stop

- Exits from the editor with an error. This is mainly for use in conjunction with **tty** commands that the user wants to abort. Since all of the commands in the editor are **errset** protected, the user must exit from the editor via a command. The **stop** command provides a way of distinguishing between a successful and unsuccessful (from the user's standpoint) editing session.

tl

- Calls (top-level). To return to the editor just use the **return** top-level command.

repack

- Permits the 'editing' of an atom or string.

(repack \$)

- Does (lc . \$) followed by repack, e.g. (repack this@).

(makefn form args n m)

- WHERE n and m are greater than zero.
- Makes (car form) an *expr* with the *n*th through *m*th elements of the current expression with each occurrence of an element of (cdr form) replaced by the corresponding element of args. The *n*th through *m*th elements are replaced by form.

(makefn form args n)

- EQUIVALENT TO: (makefn form args n n).

(s var)

- Sets var (using **setq**) to the current expression. If the current expression is a tail, the appropriate left parenthesis is generated.

(s var . \$)

- Performs the location command (lc . \$) and then sets var to the new current expression. For example, (s foo -1 1) will set foo to the first element in the last element of the current expression.





## 17 Packages

- 17.1 Introduction 17-1
  - 17.1.1 Consistency rules 17-2
- 17.2 Package names 17-2
- 17.3 Translating strings to symbols 17-3
- 17.4 Exporting and importing symbols 17-3
- 17.5 Built-in packages 17-4
- 17.6 Package system functions and variables 17-4
- 17.7 Modules 17-9
- 17.8 Creating packages in files 17-10





# 17 Packages

## 17.1 Introduction

LISP systems have traditionally had “flat” name-spaces. That is, all symbols resided in a common pool, and could be used by any program. Although there were some techniques that could be used to restrict conflicts, large LISP systems with modules written by many different programmers benefit from a mechanism for the avoidance of accidental name collisions. FRANZ LISP promotes modular programming and addresses this name-space problem through its **package system**. The implementation in FRANZ LISP is designed to conform to that of the Common LISP design as given in *Common LISP: The Language* by Guy H. Steele (Digital Press, 1984), which we refer to as *CLTL* in this chapter.

The programmer who is used to the interactive nature of LISP program development should be aware that the design of the package system in Common LISP is based on a file-centered view of program development. Attempts to resolve the meaning of the multiple-name-space scoping rules in the midst of the run-time system and interpreter have resulted in heated debate (after the publication of *CLTL*). Nevertheless, packages provide a useful facility, especially for debugged programs. It is *possible* to get substantially tangled up if the programmer is defining and modifying several inter-related packages with common names while interactively debugging and editing.

A **package** is a vector of data which establishes a mapping from external names to internal symbols. The single current package is used by the LISP reader and printer in converting lexical strings into symbols. The value of the global variable *\*package\** is bound to the current package. You can refer to symbols in other packages by the use of *package qualifiers*. A package qualifier is a string (the name of the home package for the symbol) used as a prefix to the symbol name. Usually one colon is used to separate the prefix from the symbol name. For example, you can use the name `restaurant:order` to refer to the symbolic object `order` which is defined in the `restaurant` package, if, within that package, `order` is defined to be an **external** symbol. In some programming languages, any symbols not declared external would simply not be accessible from outside the defining package. In LISP, it is merely made inconvenient. If `order` were not external, but **internal**, it could be reference by the syntax `restaurant::order`. Any symbol is either **external** or **internal** relative to a package.

External symbols are advertised outside a package to be available for use by programs in other packages. Their names should be chosen to be unique and associated with the useful abstractions of the package module you are building. On the other hand, you



should use internal symbols for implementation of abstractions which you wish to conceal from ordinary external users. By default all symbols are created as internal symbols. You explicitly use the **export** command to identify external symbols. An important consequence of this is that different packages can reuse the same internal name without conflict.

Each symbol is implemented with a **package cell** containing a pointer to its unique **home package** in which it is said to be **interned**. That is, its representation is internalized in the LISP interpreter's symbol table. It is possible and sometimes useful to create **uninterned** symbols which have *nil* in their package cell.

In using packages, you have the option of creating a hierarchy of naming by inheritance. By using the functions **intern**, **import**, and **export**, can be used to make a previously inaccessible symbols accessible in a package from other packages.

The function **unintern** makes a symbol inaccessible from a package. If the package was its home package, the symbol is said to be **uninterned**.

### 17.1.1 Consistency rules

These package consistency rules hold true so long as you do not change the value of *\*package\**:

- *Read-read consistency*: Reading the same string always results in the same (**eq**) symbol.
- *Print-read consistency*: An interned symbol always prints as a sequence of characters that, when read back in, yields the same (**eq**) symbol.
- *Print-print consistency*: If two interned symbols are not **eq**, then their printed representations will be different sequences of characters.

The rules may not hold if you change the value of *\*package\**, or doing so implicitly by continuing execution from an error break, or if you call one of the "dangerous" functions described subsequently. These functions are **unintern**, **unexport**, **shadow**, **shadowing-import**, and **unuse-package**.

## 17.2 Package names

When a package is created, it is given a character string name. Nicknames or shortened versions of the "official" name can also be assigned at that time. A number of functions described below provide mappings between these concepts (**find-package**, **package-name**, **package-nicknames**, **find-package**, **rename-package**). Any of the functions that require a package-name argument from the user accept either a symbol or a string. If the user supplies a symbol, its print name will be used. A package, if printed, will look like #<"package name" package>.



### 17.3 Translating strings to symbols

At any time there is a current package object in effect. It is the value of the global symbol *\*package\**.

The current package affects the way a string, read in by the FRANZ LISP reader, is interpreted as a symbol. Since the current package may also (and usually will) inherit names from other packages, these must be searched as well as the package currently in effect. If the string is associated with no symbol at all, a new symbol must be created and **interned** in the current package. The point of all this is to make sure that if the same string is later read in the same package context, then the the same symbol will be used.

When you wish to refer to an external symbol in a package other than the current package, you can use a *qualified name*, which is a concatenation of the package name, a colon (:), and the name of the symbol. If you need to refer to an **internal** symbol of some package other than the current one, you can use a qualified name using a double colon (::) instead of the usual single colon as the separator for the qualified name.

There is a distinguished package named *keyword* which contains all keyword symbols used by the LISP system itself and by user-written code. Because keyword symbols are used so frequently and must be accessible from everywhere, there is a special reader syntax for them which omits the package name, leaving just the colon and the symbol. Furthermore, all symbols in the keyword package are external constants and are also implicitly quoted. For example, the symbol **:foo** is the same as *keyword:foo*, and has the value **:foo**. As implemented in FRANZ LISP, each symbol contains a package cell (accessed by the function *symbol-package*) that points to the home package of the symbol. Uninterned symbols have *nil* home package pointers.

Symbols are printed in a form which would allow them to be read in and identified with themselves, assuming the same package is current. For example, keywords are printed with a preceding colon. Normally accessible symbols are printed without colon-qualifications. Symbols which are accessible only by internal or external qualification with respect to an accessible package are printed with the appropriate colons. An unin-terned symbol is printed preceded by **#:**. When read in, such a string causes an unin-terned symbol to be created.

### 17.4 Exporting and importing symbols

While the principal advantage of using packages is to allow for the creation of separate modules and name spaces, this would not be of much use without features for controlled inter-package access to names. Other than the use of the colon syntax for reference to external or internal symbols in packages, two other mechanisms are available: **import** and **use-package**.

The function **import** will not allow you to shadow (i.e. make inaccessible) a currently accessible distinct symbol either in the current package or available by inheritance. (If the same exact symbol is already present, then **import** has no effect.) If you really want to shadow a symbol you can use the function **shadowing-import**.



**use-package** makes symbols accessible by inheritance, but does not import them. It checks for name conflicts between the newly imported symbols and those already accessible in the importing package. The function **unuse-package** undoes this.

FRANZ LISP uses the standard Common LISP prescriptions for resolution of name conflicts.

## 17.5 Built-in packages

The following packages are built into every FRANZ LISP system, Opus 42 and later:

*lisp*: This package contains the standard user-visible functions and global variables that are present in the FRANZ LISP system. Unless it is the intention to hide the standard LISP functionality from the user-level program, any package should use *lisp* so that these symbols will be accessible without qualification. The nickname *fl* is available for this package.

*commonlisp*: This package, with nickname *cli* contains the primitives of the Common LISP core-compatibility package. Its external symbols include the user-visible functions and global variables that are present in the Common LISP system, including functions which (because of the incompatible re-use of the same name) must shadow the default FRANZ LISP system. The user can access the original FRANZ LISP functions by using the *fl*: prefix. Other packages based on Common LISP should use *cli* so that these symbols will be accessible without qualification.

*user*: This package is the default current package at the time a LISP system is started. This includes the usual top-level interpreter and debugger in FRANZ LISP. In order to provide the usual LISP environment, this package uses the *lisp* package. *keyword*: All the symbols in this package are treated as constants that evaluate to themselves, so that the user can type *:test* instead of *'test*. All the keywords used by built-in or user-defined LISP functions are automatically placed in this package as external symbols, and are always represented with an initial colon.

*editor*: This package contains the resident editor and its subroutines. This package uses *lisp* and has the nickname *ed*. The editor and its package are loaded by the auto-loading facility when certain of its external functions are invoked. See chapter 13 for details.

*system*: This package contains system-dependent hooks such as **getuid** (get user identification).

*system-internals*: (Nickname *si*.) This package contains programs which are FRANZ LISP system internal primitives usually not used directly by applications programmers.

*flavors*: This package contains an object-oriented programming system. See chapter 19 for details.

## 17.6 Package system functions and variables

In order to promote compatibility resulting from loading compiled or "source" files, certain package operations are treated as though they were surrounded by (eval-when (compile load eval) ...); **eval-when** is described in Chapter 12. These operations are **make-**



**package**, **in-package**, **shadow**, **shadowing-import**, **export**, **unexport**, **use-package**, **unuse-package**, and **import**. These functions should appear only at top level within a file. The recommended procedure is to write files so that all of the package set-up forms precede actual programs. Normally, then, at most one package would be defined per file. In a large package, it would be usual for several files to collectively define all the required programs.

For the functions described here, all optional arguments named **package** default to the current value of *\*package\**. Where a function takes an argument that is either a symbol or a list of symbols, an argument of *nil* is treated as an empty list of symbols.

#### *\*package\**

- The value of the variable *\*package\** must be a package (not the name of a package: see **find-package** below to see how to obtain a package from its name). The value of *\*package\** is referred to as the current package. Its value is initially the *user* package. During loading, *\*package\** is *lambda-bound* to its current value, so that it is restored automatically at the conclusion of the loading process.

**(make-package 'st\_pkname [:nicknames 'l\_names] [:use 'l\_packs])**

- RETURNS a new package with the specified package name.
- NOTE: *st\_pkname* may be either a string or a symbol. *l\_names* is a list of strings or symbols to be used as alternative names for the new package. *l\_packs* is a list of packages or the names of those packages whose external symbols are to be inherited by the package being defined. These packages must already exist. If not supplied, The default value of *:use* is a list of one package, the *lisp* package.

**(in-package 'st\_pkname [:nicknames 'l\_nicklist] [:use 'p\_usepack])**

- SIDE EFFECT: the **in-package** function is intended to be used as a declaration placed at the start of a file containing a subsystem that is to be loaded into some package other than *user*. If there is not already a package named *st\_pkname*, this creates this package as though **make-package** were called, and then sets *\*package\** to this. Although this can be changed during the loading of the file by another call to **in-package**, normally the programmer will just allow the *\*package\** variable to revert to its old value at the completion of the load operation.
- NOTE: If *st\_pkname* names any existing package, presumably this is a reloading or augmentation of a package that was loaded previously.

**(find-package 's\_name)**

- RETURNS the package with the name or nickname *s\_name*, otherwise *nil*.



(package-name 'k\_pack)

- RETURNS a string, which is the name of the package k\_pack.

(package-nicknames 'k\_pkname)

- RETURNS a list of strings that are nicknames for the package k\_pkname. If there are no nicknames, *nil*.

(rename-package 'k\_pkname 's\_newname ['l\_newnicknames])

- SIDE EFFECT: Following application of **rename-package**, the package k\_pkname is denoted by s\_newname and the nicknames, l\_newnicknames, if any. The old name and nicknames no longer access a package.

(package-use-list 'k\_pack)

- RETURNS a list of all packages used by package k\_pack.

(package-used-by-list 'k\_pack)

- RETURNS a list of other packages that use package k\_pack.

(package-shadowing-symbols 'k\_pack)

- RETURNS a list of symbols that have been declared as shadowing symbols in package k\_pack by **shadow** or **shadowing-import**. All symbols on this list are present in the specified package.

(list-all-packages)

- RETURNS a list of all packages, regardless of usage, that exist in the current LISP system.

(intern t\_string ['k\_package])

- WHERE the package k\_package (or if missing, the default package *\*package\**), is searched for a string t\_string. This search will include inherited symbols. If a symbol with the specified name is found, it is returned. If no such symbol is found, one is created and is installed in the specified package as an internal symbol (as an external symbol if the package is the *keyword* package); the specified package becomes the home package of the created symbol.

- RETURNS multiple values: the first is the symbol that was found or created, and the second is one of the following four symbols:

- nil* if t\_string is being established for the first time with this call;
- :internal* if t\_string was already in the named package as an internal symbol;
- :external* if t\_string was already in the named package as an external symbol;



*:inherited* if *t\_string* was inherited in the named package.

■ **NOTE:** **intern** may also be given a symbol (instead of a string), in which case, the symbol, if not found, will be inserted into the specified package. This allows unin-terned symbol to be given a home package.

---

```
=> (setq r (make-package 'newpack))
#<package newpack>
=> (intern "foo" r)
multiple values returned: newpack::foo nil
newpack::foo
=> (intern "foo" r)
multiple values returned: newpack::foo :internal
newpack::foo
```

---

**(find-symbol t\_string ['k\_package])**

■ **RETURNS** multiple values: *nil* and *nil* if the indicated symbol is inaccessible; otherwise a result identical to **intern**.

**(unintern 's\_symbol ['k\_package])**

■ **RETURNS** *t* if *s\_symbol* is found and removed, and *nil* otherwise.

■ **NOTE:** **unintern** changes the state of the package system and may cause problems with consistent reading and writing.

**(export 'ls\_syms ['k\_package])**

■ **WHERE** *ls\_syms* is a symbol or a list of symbols.

■ **SIDE EFFECT:** These symbols become accessible as external symbols in package *k\_package*.

■ **NOTE:** A call to **export** at the beginning of a file should be used to announce its public symbols. An error break will result if the **export** results in symbol clashes. Using the **?ret** command to **tpl** top level will cause execution to continue and the user may be prompted as to which symbols should be uninterned to avoid the clash.

■ **RETURNS** *t*.

**(unexport 'ls\_sym ['k\_package])**

■ **WHERE** *ls\_sym* is a list of symbols, or a single symbol. These symbols, which presumably were external symbols in the indicated package (or the current package if *k\_package* is not given) become internal symbols in package *k\_package*.

■ **RETURNS** *t*.



(import 'lg\_sym ['k\_package])  
(shadowing-import 'lg\_sym ['k\_package])

- NOTE: The argument `lg_sym` should be a list of symbols, or possibly a single symbol. These symbols become internal symbols in package `k_package` and can therefore be referred to without having to use qualified-name (colon) syntax. **import** signals a correctable error if any of the imported symbols has the same name as some distinct symbol already accessible in the package. **shadowing-import** plows through without the correctable error, and furthermore notes the symbols as shadowing.
- RETURNS `t`.

---

```
=> (import 'newpack::foobar)
; Attempt to import symbol foobar.
(foobar)
; A name conflict causes a break.
c{1} ?ret
; We continue and allow the imported name to shadow
; the symbol that was previously accessible.
t
```

---

(shadow 'ls\_sym [k\_package])

- SIDE EFFECT: For each symbol in the list `ls_sym`, if it is directly present in the specified package, nothing is done. Otherwise it is instantiated as an internal symbol and placed on the shadowing-symbols list of package `k_package`.
- RETURNS `t`.

(use-package 'lkst\_packs [k\_package])

- WHERE `lkst_packs` is a list of packages, a single package, a list of (string or symbol) package names, or a single package name;
- SIDE EFFECT: These packages are merged onto the use-list of `k_package`. All external symbols in the packages to use become accessible in `k_package` as internal symbols.
- RETURNS `t`.

(unuse-package 'lkst\_packs [k\_package])

- SIDE EFFECT: reverses the effect of **use-package**.
- RETURNS `t`.



**(find-all-symbols st\_name)**

- RETURNS a list of all symbols whose print name is st\_name. If a symbol is provided, its print name is used for the search.

**(do-all-symbols l\_ilst l\_body)**

- WHERE the iteration list l\_ilst has the form (s\_var [g\_result]).
- SIDE EFFECT: **do-all-symbols** provides iteration over all symbols in all packages. The l\_body is performed once for each symbol, where, each time through l\_body, the s\_var is bound to a different symbol.
- RETURNS nil or the result of evaluating the optional result-form g\_result.

**(do-symbols l\_ilst l\_body)****(do-external-symbols l\_ilst l\_body)**

- WHERE the iteration list l\_ilst has the form (s\_var [k\_package [g\_result]]).
- SIDE EFFECT: **do-symbols** provides iteration over the symbols of a package. The l\_body is performed once for each symbol accessible in the specified package, where, each time through l\_body, the s\_var is bound to a different symbol.
- RETURNS nil or the result of evaluating the optional result-form g\_result.
- NOTE: **return** may be used to terminate the iteration prematurely. The function **do-external-symbols** iterates over the external symbols only.

---

```
=> (do-symbols (i (find-package 'system) 'done)
               (progn (print i) (terpri)))
; * * *
system:link
system:time
system:int-serv
system:nice
system:unlink
system:gethostname
system:getuid
system:fpoint-serv
done
=>
```

---

## 17.7 Modules

A **module** is a Common LISP subsystem that is loaded from one or more files. A module is normally loaded as a single unit, regardless of how many files are involved. A module may consist of one package or several packages. The file-loading process is necessarily implementation-dependent, but Common LISP provides rudimentary machinery for naming modules, for keeping track of which modules have been loaded, and for loading



modules as a unit.

*\*modules\**

■ This variable is a list of names of the modules that have been loaded into the LISP system so far. This list is used by the functions **provide** and **require**. Each module has a unique name (a string). The **provide** and **require** functions accept either a string or a symbol as the module-name argument. If the module consists of a single package, it is customary for the package and module names to be the same.

(**provide** 's\_name)

■ NOTE: This declarative function is used in a file defining a set of functions.

■ SIDE EFFECT: Adds the module name s\_name to the list of modules *\*modules\**, indicating that the s\_name module has been previously loaded.

(**require** 's\_name ['sl\_pathname])

■ SIDE EFFECT: The **require** function tests whether the s\_name module is already present. If not, it proceeds to load the appropriate file or set of files. The pathname argument, if present, is a single pathname or a list of pathnames whose files are to be loaded in order, left to right. Conventional path search-names will be used if necessary to find the files.

□ A convenient way to customize a system is to require certain modules to be loaded into *user* package whenever FRANZ LISP is started afresh. Typically an installation or a group of programmers will set up an initialization file in the *.lisprc* file when FRANZ LISP for this purpose. Usually such an initialization file simply causes other facilities to be loaded.

## 17.8 Creating packages in files

When each of two files uses some symbols from the other, one must be careful to arrange the contents of the file in the proper order. Typically each file contains a single package that is a complete module. The contents of such a file should include the following items, in order:

- A call to **provide** that announces the module name.
- A call to **in-package** that establishes the package.
- A call to **shadow** that establishes any local symbols that will shadow symbols that would otherwise be inherited from packages that this package will use.
- A call to **export** that establishes all of this package's external symbols.
- Any number of calls to **require** to load other modules that the contents of this file might want to use or refer to. (Because the calls to **require** follow the calls to **in-package**, **shadow**, and **export**, it is possible for the packages that may be loaded to refer to external symbols in this package.)
- Any number of calls to **use-package**, to make external symbols from other packages accessible in this package.

- Any number of calls to **import**, to make symbols from other packages present in this package.
- Finally, the definitions making up the contents of this package/module.

For very large modules whose contents are spread over several files it is recommended that the user create the package and declare all of the shadows and external symbols in a separate file, so that this can be loaded before anything that might use symbols from this package.





## 18 The foreign-function interface

- 18.1 A simple example 18-1
- 18.2 Loading C functions 18-2
- 18.3 Foreign-function disciplines 18-4
  - 18.3.1 Call-by-address disciplines 18-4
  - 18.3.2 Call-by-value disciplines 18-5
- 18.4 Calling C functions 18-5
  - 18.4.1 Integers and real numbers 18-6
  - 18.4.2 Strings and symbols 18-6
  - 18.4.3 Vectors and arrays 18-6
  - 18.4.4 Lists 18-8
  - 18.4.5 Passing C structures by value 18-10
  - 18.4.6 Accessing C structures from LISP 18-11
    - 18.4.6.1 An example 18-11
    - 18.4.6.2 Syntax 18-13
    - 18.4.6.3 Semantics 18-14
    - 18.4.6.4 Array accessors 18-15
    - 18.4.6.5 Pointers to structures 18-15
    - 18.4.6.6 Lisp data 18-15
    - 18.4.6.7 Debugging tools 18-16
- 18.5 The C program 18-16
  - 18.5.1 External references 18-16
  - 18.5.2 Memory allocation 18-17
  - 18.5.3 Calling LISP from C 18-17
- 18.6 Fortran and Pascal 18-18
- 18.7 Pipes 18-22
  - 18.7.1 LISP in a pipeline 18-22
  - 18.7.2 Using subprocesses 18-23
  - 18.7.3 The perils of pipes 18-23





## 18 The foreign-function interface

FRANZ LISP is unusual in its capability to load object modules into a running system and to call *foreign* functions, i.e. functions and subroutines written in other programming languages. This document describes how FRANZ LISP can interface with functions written in 'C' and other similar languages such as Pascal and Fortran. The FRANZ LISP process can also interface with other processes by means of Unix pipes

### 18.1 A simple example

This section gives an example of a 'C' function which can be compiled and loaded into FRANZ LISP. It can then be called as if it were a LISP function. The example is a function of two arguments, an integer and a real, and returns an integer value.

The basic integer and real types in FRANZ LISP are called *fixnums* and *flonums*, respectively. *Fixnums* are 32-bit two's complement integers and *flonums* are 64-bit floating-point numbers. The corresponding types in 'C' are *ints* and *doubles*.

---

The size of an *int* depends on the implementation of 'C'. If *int* is not 32 bits but *long int* is, then the 'C' program must specify *long int* instead of *int*.

---

The following 'C' function takes an integer and a real argument, and returns 1 if the integer is larger, and 0 otherwise.

```
int
greater(x, y)
    int x;
    double y;
    {
        return (x > y) ? 1 : 0;
    }
```

In the Unix operating system, this function can be put into a file called *file.c* and compiled with

```
% cc -c file.c
```

The resulting object file is *file.o*. Within LISP, the 'C' object file can be loaded with:

```
(cfasl "file.o" "_greater" 'greater "c-function")
```

The function **cfasl** loads the file *file.o*, looks for an entry point called `_greater`, and makes it available as the LISP function **greater**. The string "c-function" indicates that the



---

N.B. Most Unix 'C' compilers prepend an underscore to the names of all external symbols. This is not a universal practice among 'C' compilers, however, and, for example, the VMS VAX-11 'C' compiler does not alter the names of external symbols.

---

'C' function **greater()** expects its arguments to be passed by value rather than by reference.

The 'C' function can use any of the features normally available to 'C' functions. It can read and write to the standard input, output, and error streams, it can open and close files, and it can call 'C' library functions. Although the function **greater** merely uses the

---

Although the 'C' language is used in an exemplary fashion in this chapter, what can be done from 'C' can often be done from other languages also. The methodology may differ, and the language may circumscribe what is possible, but the same functionality can often be achieved. And it should be noted that 'C' will not necessarily be the most versatile language for interfacing with LISP.

---

values passed to it and computes a value that is returned to LISP, it is possible to use foreign functions in more complicated ways. For example, foreign functions can be used for their side effects, in applications such as graphics and multiprocessing, and in interfaces to operating systems and low-level hardware support. Foreign functions can also manipulate their arguments, allocate and return objects in the LISP data space, and even call LISP functions. These possibilities are touched upon later in this chapter.

---

## 18.2 Loading C functions

As illustrated above, a function can be loaded into LISP and then invoked. A function not written in LISP must still respect the LISP environment. Usually, foreign functions are written so they can be treated as *black boxes* that take arguments and produce answers, but produce no side effects on the environment.

Before being loaded into LISP, a function must be incorporated into an object module along with all user-defined functions that it calls and all user-defined external symbols that it references. The 'C' compiler's output (e.g., *work.o*) is an object module, but the loader's usual output (e.g., *a.out*) is an executable file, not an object module. If all the functions of interest are in one file, and it is compiled with the *-c* flag, an object module will be produced.

If the functions are in more than one file and refer to each other, then the compiled object modules should be combined into a single module that can be loaded into LISP while resolving external references. The modules are combined by the loader, with its *-r* switch. The resulting file, either *a.out* or whatever follows the *-o* switch, is in this case, an object, not an executable, module.

For example, suppose we want to combine the files *in.c*, *out.c*, and *work.c* into the object module *code.o*:

```
cc -c in.c out.c work.c
ld -r in.o out.o work.o -o code.o
```



Once in LISP, you load an object module with the **cfasl** function.

**cfasling** a file is somewhat slower than loading a equivalently-sized file of LISP functions with **fasl**. If a collection of 'C' functions is commonly used (and sufficient disk space is available), the process containing the **cfasled** functions can be saved by the **dumplisp** function.

When a file is **cfasled** into LISP, there is always the risk that a name conflict will occur. If you use an external name which happens to be the same as an external name that is used by the LISP system, then the loader will report an error that the name is multiply defined. There is no way for you to change a name inside the LISP kernel, so you must change the entry-point name in the file that you are **cfasling** in.

```
(cfasl 'st_fname 'st_cname 's_lispname ['st_discipline ['st_libraries]])
(ffasl 'st_fname 'st_cname 's_lispname ['st_discipline ['st_libraries]])
```

■ WHERE **st\_fname** is the module's file name, **st\_cname** is the entry point of the 'C' function name; **s\_lispname** is the LISP symbol name to bind to the 'C' function, and can be different from the entry-point name; **st\_discipline** is function's *calling discipline* (see below); **st\_libraries** is also optional and lists the libraries to be searched to resolve references in the object module. The form of **st\_libraries** follows that of the native operating system loader. For example on Unix, one would use the string *"-lc"* to load the 'C' run-time library.

■ RETURNS *t* if the load was successful; *nil* otherwise.

■ SIDE EFFECT: The file **st\_fname** is loaded, and the function **s\_lispname** is made available in the LISP environment. **cfasl** normally prints out the command line passed to the loader on *poport*. This message can be suppressed by binding the special symbol *\$ldprint* to *nil*.

□ **ffasl** is just like **cfasl** except that it is intended for loading Fortran modules. The **cfasl** function loads the 'C' run-time library automatically. The **ffasl** function loads the Fortran run-time libraries automatically.

■ NOTE: Some 'C' compilers prepend an underscore (*\_*) to external symbols. The entry-point name given to **cfasl** or **ffasl** must be the name known to the linker, which may not be identical to the name of the function, subroutine, or procedure in the source language.

□ The **cfasl** and **ffasl** functions only define one entry-point name to LISP. If the object module being loaded has more entry points of interest, they must be defined to LISP using the **getaddress** function described below. Also, a file once loaded with **cfasl** or **ffasl** may not be loaded again without first removing the entry-point names from the LISP environment with the **removeaddress** function, also described below.

```
(getaddress 'st_cname 's_lispname ['st_discipline] ...)
```

■ WHERE **st\_cname** is the entry-point name (as known to the native operating-system linker), **s\_lispname** is the LISP symbol to bind to the binary object corresponding to the foreign-function entry point, and **st\_discipline** is the optional discipline of the function.



- RETURNS the binary LISP object bound to the symbol `s_lispname` if the entry point `st_cname` is found; *nil* otherwise.
- SIDE EFFECT: The symbol `s_lispname` is bound to the binary object representing the foreign function.
- NOTE: To make several functions available, it is more efficient to call **getaddress** once with a sequence of triples as arguments than to call **getaddress** several times.

(**removeaddress** 'st\_cfunc ...)

- WHERE `st_cfunc` is the name of a foreign-function entry point.
- RETURNS *t* if the external symbol `st_cfunc` was found and successfully removed from the LISP environment; *nil* otherwise.
- SIDE EFFECT: The function **removeaddress**, removes the name of a foreign function from the LISP environment. It must be used prior to loading a file that contains an external symbol already in the current LISP environment.

### 18.3 Foreign-function disciplines

Every foreign function has a **discipline** that is declared when the function is incorporated into LISP. The **discipline** tells LISP how the function interprets its arguments and the type of value returned, if any.

Franz LISP recognizes eight foreign-function disciplines. The first four are more suited for *call by address* languages, like Fortran, while the last four are designed for *call by value* languages like 'C'.

#### 18.3.1 Call-by-address disciplines

"function"

- This is a function which takes LISP values as arguments, and returns a LISP value. This is as close to a LISP function as a foreign function can be. In LISP function arguments are normally evaluated, but a LISP value is actually a pointer to a data structure. Thus the function must return a **pointer** to a valid LISP object, or a LISP error may result. Normally this is accomplished by passing the function a LISP object that is then modified and returned as the value of the function. Since the resulting value originated in LISP, LISP treats it like any other LISP object.

"subroutine"

- This is the default discipline. As with the "function" discipline, the arguments passed to the function are pointers to LISP objects. However, with a subroutine, the return value, if any, is ignored. FRANZ LISP always returns *t* from a subroutine.



*"integer-function"*

- This returns an integer which FRANZ LISP stores as a *fixnum*. Again the arguments passed to the function are pointers to LISP objects.

*"real-function"*

- This is just like an *"integer-function"* except that the value returned is a double-precision real number that FRANZ LISP stores as a *flonum*.

### 18.3.2 Call-by-value disciplines

*"c-function"*

- This is a 'C' function which returns an integer, which FRANZ LISP stores as a *fixnum*. This differs from an *"integer-function"* in that *fixnum* and *flonum* arguments are passed by value. Other arguments are passed unchanged.

*"void-c-function"*

- This is a 'C' function whose return value is ignored. It is just like *"c-function"* except that the LISP function returns *t*. (In 'C', the function can be declared to be of type **void**, or of any other type.)

*"double-c-function"*

- This is a 'C' function that returns a *double*, which FRANZ LISP stores as a *flonum*. Arguments are treated the same as for a *"c-function"*.

*"vector-c-function"*

- This is a 'C' function that returns a structure. This type of function is used less frequently and is explained in a later section.

## 18.4 Calling C functions

The major requirement for successfully using foreign functions is an understanding of how values are passed to the functions and how the results are returned. LISP has a definite conception of where particular types of objects belong and of their structure. 'C', on the other hand, respects fewer conventions, and 'C' functions have difficulty in using the intrinsic LISP functions for creating, modifying, or storing LISP objects. Foreign functions can find the fields of LISP structures and use the values in them, but these functions cannot easily deduce the types of LISP objects. Hence arguments to foreign functions and values returned are assumed to be of known correct types. Values created and returned by foreign functions are either treated as LISP values, without any checking, or as tightly closed *black boxes*.



LISP values passed to foreign functions are treated specially by LISP, with the exact treatment depending on the value's type.

### 18.4.1 Integers and real numbers

Integers and real numbers are copied and a pointer to the new value is passed. The values are copied so that foreign functions cannot change small *fixnums*, which exist in pure space. Arguments of this sort cannot be used for returning results to LISP.

### 18.4.2 Strings and symbols

Strings are passed as pointers to NUL-terminated sequences of characters. Changes made by the 'C' function to a string affect the original LISP string.

Symbols are passed as pointers to the symbol's data structure (see Chapter 1). The symbol's data structure has many useful fields, but the whole structure generally need not be passed to a foreign function. If the foreign function needs only the symbol's literal representation, use its print name (see `get_pname`), which is a string easily handled by 'C' functions.

### 18.4.3 Vectors and arrays

Vectors, immediate vectors, and arrays may be created in LISP and passed to 'C' functions as arguments.

LISP immediate vectors are similar to one-dimensional arrays in 'C'. A byte immediate vector (created by `vectori-byte`) corresponds to a 'C' `char` array, a word immediate vector corresponds to a 'C' `short int` array, a longword immediate vector corresponds to a 'C' array of `long int`, a float immediate vector corresponds to a `float` array, and a double immediate vector corresponds to a `double` array.

The following 'C' function fills a given array with zeros:

```
int
zero(n, a)
    int n, a[];
{
    int i;

    for (i = 0; i < n; i++)
        a[i] = 0;

    return n;
}
```

If this is in a file called `zero.c` and compiled into an object file called `zero.o`, then it can be loaded into LISP and called by LISP as follows:

---

```
=> (cfasl "zero.o" "_zero" 'zero "c-function")
t
=> (setq x (new-vectori-long 10))
#vector[40]
=> (vseti-long x 0 27) ;; x[0] = 27
27
=> (vrefi-long x 0) ;; print it out
27
=> (zero 10 x)
10
=> (vrefi-long x 0) ;; now it's gone
0
=>
```

---

A LISP vector is similar to an array of pointers in 'C'. The following 'C' function rotates the pointers in a given array:

```
int
rotate(n, a)
    int n, *a[];
{
    int **p, **q, **r, *t;
    t = *a;
    for (p = a, q = a + 1, r = a + n; q < r; p++, q++)
        *p = *q;
    *--r = t;
    return n;
}
```

If this is in a file called *rotate.c* and compiled into an object file called *rotate.o*, then it can be loaded into LISP and called by LISP as follows:

---

```
=> (cfasl "rotate.o" "_rotate" 'rotate "c-function")
t
=> (setq x (vector 0 1 2 3))
#vector[4]
=> (rotate 4 x)
4
=> (vref x 0)
1
=> (vref x 1)
2
=> (vref x 2)
3
=> (vref x 3)
0
=>
```



Arrays can be passed to foreign functions, but they are somewhat more complicated than immediate vectors and in most cases it is preferable to use immediate vectors. For the *call-by-address* discipline, arrays are passed as pointers to an array's *data* field, which is a sequence of pointers to data. For the *call-by-value* discipline, arrays are passed as pointers to the array data structure. The appropriate structure is defined in the header file *lstructs.h* which is part of the FRANZ LISP distribution.

The function below will clear the elements of the *fixnum* array passed to it.

```
#include "global.h"
int
clear_array(a)
    struct array *a;
{
    long int **l = (long int **)a->data;
    long int **e;
    extern long int etext;

    if (a->delta->i != 4) return 0;

    for (e = l + a->length->i; l < e; l++)
        if (-1023 <= **l && **l <= 1024)
            fprintf(stderr, "pure fixnum not cleared\n");
        else
            **l = 0;

    return 1;
}
```

Note that the function above attempts to forestall attempts to modify small *fixnums* stored in pure space. The file *global.h* brings in a number of internal definitions for LISP, e.g. for *struct array* in this case.

*fixnum-block* and *flonum-block* arrays are alternatives to immediate vectors for passing numbers to Fortran routines and other foreign routines using *call-by-address* disciplines. *fixnum-block* arrays are integer arrays and *flonum-block* arrays are double-precision arrays. If you want to pass a contiguous block of objects in an array, then the type of the array must be declared to be *fixnum-block* or *flonum-block*, depending on the type of LISP values that are stored in the array. (*fixnum-block* and *flonum-block* arrays are alternatives to immediate vectors for passing numbers to Fortran routines.) Changing a value in an array propagates the new value back to LISP.

#### 18.4.4 Lists

It is possible to pass lists and other LISP objects to a 'C' function, and have the 'C' function perform LISP-like operations on them. Creating new LISP data is somewhat tricky, and recommended only for sophisticated FRANZ LISP users.



As an example of some of the pitfalls, consider a 'C' function intending to return a LISP list containing two integers. The 'C' function might require that it be passed a workspace of a LISP list of length two as an argument and then proceed to modify it. However, if the original list contains an integer between  $-1024$  and  $1023$ , then the 'C' function cannot change the value of that integer since it is stored in read-only memory. If the numbers are larger, then they are not stored uniquely and so can be changed.

The FRANZ LISP files *lstructs.h* and *global.h* define most of the relevant structures. It is usually kept in the directory */usr/franz/franz/h*. In particular, lists are defined there to be pointers to a data structure made of a pair of pointers (**cdr** followed by **car**, both full 32-bit words).

The following 'C' function is a simple example that manipulates lists. It takes two arguments, both lists, and destructively appends the second one to the first.

```
#include "global.h"

lispval
nconc(a, b)
{
    lispval a, b;
    lispval x, y;

    if (TYPE(a) != DTPR || TYPE(b) != DTPR) {
        printf("Error: arguments must be lists.\n");
        return nil;
    }

    if (a == b) return b;

    for (x = a; x != nil; x = x->d.cdr)
        y = x;

    y->d.cdr = b;
    return a;
}
```

The header file *global.h* must be included, as essential types and macros are defined there. This header file includes other header files in the same directory as *global.h*. In case these files are in the directory */usr/franz/franz/h* and the above 'C' function is in a file called *nconc.c*, the Unix operating system command to compile the file is

```
cc -c -I/usr/franz/franz/h nconc.c
```

A lispval (LISP value) is the type by means of which *s-expressions* are referenced in 'C'. It is a pointer to a lispobj (LISP object) which is a cell that might hold a number of different things. If the LISP object is a dotted pair, it holds two lispvals, its **cdr** followed by its **car**. In 'C', these declarations are:

```
struct atom { lispval clb; ... };
typedef union lispobj *lispval;
```



```
struct dtptr { lispval cdr, car;};
union lispobj {
    struct atom a;
    FILE *p;
    struct dtptr d;
    double r;
    ... };;
```

These structures are occasionally modified with new releases of FRANZ LISP so *including* the header file is preferable to copying the type definition.

TYPE is a macro for testing types. Lists have the type *DTPR* (dotted pair). Other type macros are in the file *ltypes.h*.

The macro *nil* is defined in *global.h*. It is a pointer to the atom called *nilatom* in 'C'. The macro *nil* is defined to be zero on some machines, but not all.

The above function **nconc()** is destructive in the sense that it usually modifies its first argument, as does the FRANZ LISP **nconc**. No storage is allocated or released.

### 18.4.5 Passing C structures by value

The 'C' language has the feature that **structures** and **unions** can be passed by value, either as an argument to a function or as the value returned by a function. FRANZ LISP can interface with a 'C' function which passes as an argument a structure if:

- 1 The function's discipline is "c-function", "void-c-function", "double-c-function", or "vector-c-function".
- 2 The structure is represented as an immediate vector in LISP.
- 3 When the structure is passed by value, its property field is set to *value-structure-argument*.

FRANZ LISP can interface with a 'C' function which returns a structure by value if:

- 1 The function's discipline is "vector-c-function".
- 2 When the 'C' function is called from LISP, the first argument in LISP is an immediate vector into which the result is to be stored. The second LISP argument is then the first argument to the 'C' function, and so on.

An example where structures are passed by value both to and from the 'C' function is the following function:

```
struct foo {
    int a, b, c;
};

struct foo
bar(x)
{
    struct foo x;

    int t;

    t = x.a;
    x.a = x.b;
```



```

    x.b = t;
    x.c += 13;

    return x;
}

```

If this function is in a file called *barfile.c* and compiled into *barfile.o*, then it can be loaded into LISP with:

---

```

=> (cfasl "barfile.o" "_bar" 'bar "vector-c-function")
t
=> (setq foo-in (vectori-long 12 23 34)) vectori[12]
=> (setq foo-out (new-vectori-long 3))
vectori[12]
=> (vsetprop foo-in 'value-structure-argument)
value-structure-argument
=> (bar foo-out foo-in)
vectori[12]
=> (vrefi-long foo-out 0)
23
=> (vrefi-long foo-out 1)
12
=> (vrefi-long foo-out 2)
47
=>

```

---

#### 18.4.6 Accessing C structures from LISP

The file *cstruct.l* contains LISP functions which are useful for accessing 'C' structures passed back to FRANZ LISP. It is a library file and it is located (along with its compilation) with the other LISP library files (usually */usr/lib/lisp*). If you invoke the LISP function **c-declare** it will be loaded into memory by the **autoload** feature.

'C'-like structures are implemented in LISP with immediate vectors as the underlying data type. You can avoid structures in LISP just as you can avoid it in 'C' by counting byte offsets, but in either case the process is tedious and error-prone.

##### 18.4.6.1 An example

Here are some structures taken from a popular window system:

```

struct pr_pos { int x,y; };
struct pr_subregion {
    struct pixrect *pr;
    struct pr_pos pos;
    struct pr_size { int x, y; } size;
};

```



Suppose we want to call a function in 'C' which takes, as an argument, a pointer to an object of type `pr_subregion`, and assigns values to the fields of `p`. We could allocate an immediate vector of size 20, and bind it to something, say,

```
(setq p (new-vectori 20)).
```

We could then call the 'C' function with `p` as a parameter, and the 'C' function would have no trouble assigning values to the fields of `p`. However, it would be difficult to decompose the structure in LISP and access the various fields of `p`. Furthermore, the garbage collector would not handle the pointers correctly.

The above structures can be declared in FRANZ LISP in order to circumvent these difficulties. The declarations are:

```
(c-declare
 (struct pr_pos
  (x int)
  (y int)))
```

```
(c-declare
 (struct pr_subregion
  (pr * (struct pixrect))
  (pos (struct pr_pos))
  (size (struct pr_size
         (x int)
         (y int)))))
```

These declarations create a number of LISP functions. They create `make-pr_subregion` which allocates the necessary memory for the structure `pr_subregion`. They also create functions which access the fields of the structure. The LISP statement

```
(setq p (make-pr_subregion))
```

allocates memory and makes `p` a pointer to it. In LISP, `p` has the type *vectori*. If `p` is passed to 'C', it should have the 'C' declaration

```
struct pr_subregion *p;
```

The field which would be accessed in 'C' by

```
p->pos.x
```

is accessed in LISP by

```
(pr_subregion->pos.x p)
```

and assigned to the value of `b` by

```
(setf (pr_subregion->pos.x p) b)
```



### 18.4.6.2 Syntax

(c-declare l\_struct1 [ l\_struct2 ...])

- Each structure declaration `l_structi` is a list. The first element is either of the symbols *struct* or *union*. The second element is a symbol, taken to be the name of the structure. Each subsequent element is another list taken to a declaration of a member of the structure. The first element of the member declaration is its formal name and the remaining elements in the list are its type description.
- The type description may be a symbol representing a scalar type. Scalar types from 'C' that are currently implemented are *char*, *short*, *long*, *int*, *float*, *double*, *unsigned-char*, *unsigned-short*, *unsigned-int*, and *unsigned-long*. The types *int* and *long* are synonymous.
- Another entry may be the symbol *\**, indicating the type is a pointer to an object whose type is given by the remainder of the list. Pointers can be to any type, including *unions* and *structures*.
- An entry may be the symbol *function*, indicating the type is a 'C' function returning an object described by the remainder of the list. It will always be preceded by *\** since arrays or structures cannot contain functions directly. There is no facility in LISP for doing anything with a pointer to a 'C' function, except to pass it to another 'C' function which could make use of it.
- Entries may be lists. To describe an array of objects, the entry is a list beginning with the symbol *array* and the remaining entries are the dimensions of the array.
- You may indicate structures or unions either by having a list whose first element is either *union* or *struct* and whose second element identifies the structure. As in the language 'C', you can formally declare the structure here.
- The type *lispval* is provided for storing LISP data in a structure. In 'C', such an object is always a pointer, whereas in LISP it may be any LISP object.
- Here is a more complex example. First in 'C'.

```
typedef struct { int *cdr, *car;} *lispval;
struct a {
    int m;
    short n;
    struct c *p;
    struct box { int x, *y;} b;
    int (*f[3][4])();
    char c, c1, c2;
    lispval l;
    float r[3][4];
};
struct c {
    int x, y;
    lispval ls;
};
```



Then in LISP.

```
(c-declare
 (struct a
  (m int)
  (n short)
  (p * (struct c))
  (b (struct box (x int) (y * int)))
  (f (array 3 4) * function int)
  (c char)
  (c1 char)
  (c2 char)
  (l lispval)
  (r (array 3 4) float))
 (struct c (x int) (y int) (ls lispval)))
```

A structure `s` which is declared in 'C' as

```
struct a *s;
```

has a `float` entry which is accessed in 'C' as

```
s->r[1][2];
```

and in LISP as

```
(a->r s 1 2).
```

### 18.4.6.3 Semantics

By writing one of these declarations, many functions may be generated. For each structure, you will get a creation function whose name is the concatenation of **make-** and the structure name. It is actually a macro and allows you to initialize individual fields by member name. Initializers can be inserted as pairs of an unevaluated member name and an expression to compute its value. Initialization for arrays is not currently implemented. The LISP statement

```
(setq p (make-pr_pos x 50 y (+ 20 (* 5 40))))
```

is thus equivalent to the 'C' statement

```
struct pr_pos p = { 50, 20 + 5*40 };
```

The accessor functions for structure members accept both pointers to structures or unions (declared `* (struct ...)`) and structures or unions (declared `(struct ...)`) as arguments. There are four sets of accessor functions generated for each structure and its members. Their functionality is identical, and the presence of four distinct sets of names is a legacy of earlier versions of the **c-declare** macro. One set of accessor functions are by concatenating the structure name, the symbol `->`, and the member name. Another set of accessor functions is also created with structure name and member name separated by `..`. Their functionality is identical. If the member is a structure in its own right, additional



accessor functions are generated for each member of the substructure using . separating elements. Thus the LISP expression

```
(pr_subregion->pos.y p)
```

is analogous to the 'C' expression

```
p->pos.y.
```

#### 18.4.6.4 Array accessors

If a member is an array, the corresponding accessor is a function of several arguments: the base structure and as many indices as were declared. It returns an element of the array.

#### 18.4.6.5 Pointers to structures

One potentially confusing aspect of interfacing LISP to 'C' is that LISP generally uses one more level of indirection than 'C' does. In LISP, an integer is really a pointer to an integer. Likewise, the structures and pointers to structures in 'C' become pointers to structures and pointers to pointers to structures in LISP. Thus, when a structure contains a pointer to another structure, and that pointer is accessed in LISP, it is really a pointer to a pointer to a structure. The function **maknum** is useful in this context, since it effectively creates a pointer to its argument. The following example illustrates using pointers to structures.

```
(c-declare
  (struct outer
    (x int)
    (pt * (struct inner
      (a short)
      (b short))))))

(setq s (make-outer x 7))
(setf (outer->pt s) (maknum (make-inner a 12 b 23)))
(msg (*inner->b (outer->pt s)) " should be 23" N)
```

#### 18.4.6.6 Lisp data

Putting LISP data into a 'C' structure presents a garbage collection problem. The garbage collector assumes that the contents of an immediate vector is raw data, and does not pay any attention to what pointers within the immediate vector point to. For this reason, there are some special procedures for protecting LISP data in a 'C' structure from garbage collection.

When a 'C' structure has a field of type **lispval** in it, and **setf** is used with the accessor functions to assign a LISP object to that field, then a copy is put on the property list of the



immediate vector. Likewise, if a *lispval* field of a new structure is being initialized, a copy is also put on the property list.

If LISP data is put into a 'C' structure by a 'C' function, or by a LISP function other than **setf**, a copy of that LISP data must be kept elsewhere in order to protect it from garbage collection. Each **c-declare** of a structure which contains *lispvals* creates a function with the name **protect-** concatenated with the structure name. This function takes one argument, a immediate vector which has been created by **c-declare**, and copies all of the *lispval* fields within the structure into the property list of the immediate vector. For example, (**protect-slop**) will save the *lispval* fields in a immediate vector created by (**make-slop**).

#### 18.4.6.7 Debugging tools

(**describe-cs** v\_item)

- WHERE v-item is a immediate vector which has been created by a **make-** function.
- SIDE EFFECT: Information about the structure is printed out, including the structure tag, the field names, the 'C' type of each field, and the current value of each field. The list of accessor macros is also printed out.

(**double-to-float** 'f\_flo)

- RETURNS a *fixnum*, which may be passed to a 'C' routine expecting an argument of type *float*.
- NOTE: In FRANZ LISP, *flonums* are double precision floating point numbers which correspond to the 'C' type *double*. This function and **float-to-double** provide a way to convert between a LISP *flonum* (a 'C' *double*) and a 'C' *float* (which is the same size as a LISP *fixnum*).

(**float-to-double** 'x\_fix)

- RETURNS a *flonum*, after converting the *fixnum* x\_fix from a 'C' *float* to a LISP *flonum*.
- NOTE: See the discussion at **double-to-float**.

### 18.5 The C program

#### 18.5.1 External references

Foreign functions can do much of what they would do if executed autonomously. They can use and change *extern* and *static* variables and invoke other functions that are in *cfasled* files or are part of FRANZ LISP. Even input and output from foreign functions is fine.



### 18.5.2 Memory allocation

'C' functions can use the 'C' library function **malloc()** to dynamically allocate memory. Memory allocated in this way will not be garbage-collected by LISP. The LISP system knows that this value is foreign and does not allow it to be used in place of a LISP value. The only use of these foreign values is as arguments to foreign functions. However, LISP objects cannot be allocated this simply, because the LISP system must know the data type at memory allocation time. Allocating LISP objects from your own foreign function is complicated, and not recommended. It is usually much better to allocate LISP objects in LISP, and pass pointers to the 'C' function.

### 18.5.3 Calling LISP from C

LISP functions cannot get the value of a 'C' variable unless it is returned by a 'C' function. However, 'C' functions can call the built-in function **matom**, which takes a symbol's name (as a string) as its argument and returns a pointer to the symbol's data structure. If the symbol does not exist, it is created. The 'C' code can look at the symbol's value field to find a variable's binding. Usually, passing this value as an argument is safer and easier than finding it from the 'C' code.

'C' functions can invoke LISP functions. The built-in function **ftolisp()**'s first argument is an integer indicating how many arguments follow. The next argument is the symbol whose function binding is to be invoked. The rest of the arguments to **ftolisp()** are LISP values that are passed to the LISP function. **ftolisp()** returns the result returned by the LISP function. **ftolisp()** has the same arguments and effect as the **funcall** function does in LISP.

For example, suppose a 'C' function is set up to find the time consumed by LISP. The following 'C' code invokes the LISP function **ptime**:

```
x = ftolisp(1, matom("ptime"));
```

**ftolisp()** returns a value of type **lispval**, which in this case is a pointer to a *dtptr* cell.

Passing arguments to LISP from 'C' requires some knowledge of FRANZ LISP internals. For example, the following function returns the product of its arguments by calling the function **times**.

```
prod(x, y)
  int x, y;
{
  /* As long as there is no overflow, the next line produces
     the same result as return x * y; */
  return * (int *) ftolisp(3, matom("times"), inewint(x), inewint(y));
}
```

In LISP, integers are *boxed* in the sense that a LISP value of type *fixnum* is really a pointer to a memory location where the integer is stored. The built-in function **inewint()** takes an integer argument and returns a corresponding LISP value, i.e. a valid LISP pointer to a memory location where the integer is stored. **ftolisp()** returns a LISP value



which in this case is a *fixnum*, and hence a pointer to what 'C' would understand as an integer.

On some systems, notably VAXen and Suns, the function `ftolsp_()` is available. It allows the number of arguments to be omitted, so the above example is equivalent to:

```
x = ftolsp_(matom("ptime));
```

`ftolsp_()` thus has the same arguments and effect as the `funcall` function does in LISP.

## 18.6 Fortran and Pascal

The `cfasl` function loads into LISP an object file which has the format specified by the 'C' compiler. On those systems where Fortran and Pascal use the same object-module format as 'C', routines written in these languages may be loaded into LISP much the same way 'C' functions can be. The differences are:

- *Calling conventions.* 'C' is call by value, Fortran is call by address, and Pascal allows either.
- Fortran does not have pointer types.
- Pascal allows passing arrays and structures by value, but interfacing such a procedure with FRANZ LISP is different.
- Libraries must be explicitly invoked for Fortran and Pascal.

The method a foreign function uses to access the arguments provided by LISP is dependent on the language of the foreign function. The following scripts demonstrate how how LISP can interact not only with 'C', as previously discussed, but with two additional languages: Pascal and Fortran.

'C' and Pascal have pointer types and the first script shows how to use pointers to extract information from LISP objects. There are two functions defined for each language. The first (`cfoo()` in 'C', `pfoo()` in Pascal) is given four arguments, a *fixnum*, a *flonum-block* array, a *hunk* of at least two *fixnums* and a list of at least two *fixnums*. To demonstrate that the values were passed, each function prints its arguments (or parts of them). The function then modifies the second element of the *flonum-block* array and returns a 3 to LISP. The second function (`cmemq()` in 'C', `pmemq()` in Pascal) acts just like the LISP `memq` function (except it will not work for *fixnums* whereas the LISP `memq` will work for small *fixnums*).

---

```
# These are the 'C'-coded functions.
% cat ffdemo.c
/* demonstration of c coded foreign integer-function */
/* the following will be used to extract fixnums out of a list of fix-
nums */
struct listoffixnumscell {
    struct listoffixnumscell *cdr;
    int *fixnum;
};

struct listcell {
```



```

    struct listcell *cdr;
    int car;
};

int
cfoo(a, b, c, d)
    int *a;
    double b[];
    int *c[];
    struct listoffixnumscell *d;
{
    printf("a: %d, b[0]: %f, b[1]: %f0, *a, b[0], b[1]);
    printf(" c (first): %d, c (second): %d0, *c[0], *c[1]);
    printf(" ( %d %d ... )0, *(d->fixnum), *(d->cdr->fixnum));
    b[1] = 3.1415926;
    return(3);
}

struct listcell *
cmemq(element, list)
    int element;
    struct listcell *list;
{
    for(; (list != nil) && element != list->car; list = list->cdr);
    return(list);
}

# These are the Pascal-coded functions.
% cat ffdemo.p
type pinteger = ^integer;
    realarray = array[0..10] of real;
    pintarray = array[0..10] of pinteger;
    listoffixnumscell = record
        cdr : ^listoffixnumscell;
        fixnum : pinteger;
    end;
    plistcell = ^listcell;
    listcell = record
        cdr : plistcell;
        car : integer;
    end;

function pfoo(var a : integer;
               var b : realarray;
               var c : pintarray;
               var d : listoffixnumscell) : integer;
begin
    writeln(' a:', 'a, ' b[0]:, ' b[0], ' b[1]:, ' b[1]);
    writeln(' c (first):, ' c[0]^, ' c (second):, ' c[1]^);
    writeln(' ( , ' d.fixnum^, d.cdr^.fixnum^, .'...' ');
    b[1] := 3.1415926;
    pfoo := 3
end;

```



```
{
The function pmemq looks for the Lisp pointer given as
the first argument in the list pointed to by the second
argument. Note that we declare "a : integer" instead of
"var a : integer" since we are interested in the pointer
value instead of what it points to (which could be any
Lisp object).
}
function pmemq(a : integer;
               list : plistcell) : plistcell;
begin
  while (list <> nil) and (list^.car <> a) do list := list^.cdr;
  pmemq := list;
end;

# The files are compiled.
% cc -c ffdemo.c
1.0u 1.2s 0:15 14% 30+39k 33+20io 147pf+0w
% pc -c ffdemo.p
3.0u 1.7s 0:37 12% 27+32k 53+32io 143pf+0w
% lisp
Franz Lisp, Opus 43.1
; First the files are loaded and we set up one foreign
; function binary. We have two functions in each file
; so we must choose one to tell cfasl about. The choice
; is arbitrary.
=> (cfasl "ffdemo.o" "_cfoo" 'cfoo "integer-function")
/usr/lib/lisp/nld -N -A /usr/local/LISP -T 63000 ffdemo.o -e _cfoo -o
/tmp/Li7055.0 -lc
t
=> (cfasl "ffdemo.o" "_pfoo" 'pfoo "integer-function" "-lpc")
/usr/lib/lisp/nld -N -A /tmp/Li7055.0 -T 63200 ffdemo.o -e _pfoo -o
/tmp/Li7055.1 -lpc -lc
t
; Here we set up the other foreign function binary objects.
=> (getaddress "_cmemq" 'cmemq "function"
           "_pmemq" 'pmemq "function")
#<bcd 0xFA400 "c-function">
; We want to create and initialize an array to pass to
; the cfoo function. In this case we create an unnamed
; array and store it in the value cell of testarr.
; When we create an array to pass to the Pascal program
; we will use a named array just to demonstrate the
; different way that named and unnamed arrays are
; created and accessed.
=> (setq testarr (array nil flonum-block 2))
array[2]
=> (store (funcall testarr 0) 1.234)
1.234
=> (store (funcall testarr 1) 5.678)
5.678
=> (cfoo 385 testarr (hunk 10 11 13 14) '(15 16 17))
a: 385, b[0]: 1.234000, b[1]: 5.678000
```



```

c (first): 10, c (second): 11
( 15 16 ... )
3
; Note that cfoo has returned 3 as it should.
; It also had the side effect of changing the second
; value of the array to 3.1415926 which we check next.
=> (funcall testarr 1)
3.1415926
; In preparation for calling pfoo we create an array.
=> (array test flonum-block 2)
array[2]
=> (store (test 0) 1.234)
1.234
=> (store (test 1) 5.678)
5.678
=> (pfoo 385 (getd 'test) (hunk 10 11 13 14) '(15 16 17))
a:          385 b[0]:  1.234000000000000E+00 b[1]:  5.678000000000000E+00
c (first):   10 c (second): 11
(           15           16 ...)
3
=> (test 1)
3.1415926
; Now to test out the memq's.
=> (cmemq 'a '(b c a d e f))
(a d e f)
=> (pmemq 'e '(a d f g a x))
nil

```

---

The Fortran example is much shorter since in Fortran you cannot follow pointers as you can in other languages. The Fortran function `ffoo` is given three arguments: a *fixnum*, a *fixnum-block* array and a *flonum*. These arguments are printed out to verify that they made it and then the first value of the array is modified. The function returns a *double-precision* value which is converted to a *flonum* by LISP and printed. Note that the entry point corresponding to the Fortran function `ffoo` is `_ffoo_` as opposed to the 'C' and Pascal convention of preceding the name with an underscore.<sup>1</sup>

---

```

% cat ffdemo.f
double precision function ffoo(a, b, c)
integer a, b(10)
double precision c
write (*, 100) a, b(1), b(2), c
100 format(' a=,',i4,', b(1)=,',i5,', b(2)=,',i5,', c=,',f6.4)
b(1) = 22
ffoo = 1.23456
return

```

<sup>1</sup>This convention is observed on some Unix systems. Other operating systems may observe other conventions.



```

        end
% f77 -c ffdemo.f
ffdemo.f:
    ffoo:
0.9u 1.8s 0:12 22% 20+22k 54+48io 158pf+0w
% lisp
Franz Lisp, Opus 43.1
(C) Copyright 1985,1986,1987 Franz Inc., Alameda Ca.
=> (cfasl "ffdemo.o" "_ffoo_" 'ffoo "real-function" "-lI77 -lF77")
/usr/lib/lisp/nld -N -A /usr/local/LISP -T 63000 ffdemo.o -e _ffoo_
-o /tmp/Lil1066.0 -lF77 -lI77 -lc
t
=> (array test fixnum-block 2)
array[2]
=> (store (test 0) 10)
10
=> (store (test 1) 11)
11
=> (ffoo 385 (getd 'test) 5.678)
a= 385, b(1)= 10, b(2)= 11 c=5.6780
1.234559893608093
=> (test 0)
22

```

---

## 18.7 Pipes

Pipes are the principal method of program composition in the Unix operating system. They offer the considerable virtues of being easy to understand, quick to program, and standard throughout Unix. However pipes also consume substantial overhead in comparison to function calls described earlier in this chapter, and are ill-suited to passing structured data. Nevertheless, pipes are important, particularly since they are the only method used for many existing programs.

### 18.7.1 LISP in a pipeline

Frequently, a program—e.g. the shell—creates a pipeline, one of whose component programs is written in LISP. The LISP program reads its data from the standard input and writes its results to the standard output (i.e. neither `read` nor `print` requires a port be specified as an argument).

At this point, a quick reader may wonder how a LISP program, which is usually run by an interpreter, can be executed by the shell. If the LISP program is interpreted, the commands that invoke `lisp`, load the requisite files, and start the program running can be put in a shell command file, which can be invoked in the same way as any other Unix command. If the LISP program is compiled, it can be compiled in a way that allows it to execute without being explicitly loaded into the interpreter (see the `liszt` compiler's `-r` switch). The executable file produced by `liszt` with the `-r` flag is invoked as is any other



compiled program; but before the LISP code is run, the LISP interpreter is invoked and the compiled code loaded.

### 18.7.2 Using subprocesses

The complementary situation is not as simple—a LISP program invokes another program, perhaps supplying it with data and retrieving its results. In addition to starting this program running, someone must connect the pipes that service the program. LISP provides functions to handle this.

Releases of FRANZ LISP after Opus 38.30 have a very flexible set of functions—**\*process**, **\*process-send**, and **\*process-receive**.

**\*process** takes between one and three arguments. The first is the subprocess's file name. The other two are optional boolean flags that, if present and non-*nil*, cause **\*process** to open ports from and to the subprocess, respectively. **\*process** returns a list of up to three items. If ports are opened, they are the first items. If a port is not opened, nothing is put into its place in the list (i.e. if only one port is opened, the list contains two items). The last item in the list is always the subprocess's id number. The values in the list are easily assigned to individual variables by the **desetq** function.

The example above reduces to:

```
(let*
  ((ResultPort (car (*process 'pwd nil t)))
   ; NB args evaluated
   (Result (read ResultPort)))
  (close ResultPort)
  Result)
```

Since communication with a process is frequently in one direction and the order of arguments to **\*process** is difficult to remember, the other two functions **\*process-send** and **\*process-receive** take only the subprocess's name and return a list of an open port and a process id.

Hence the example is best written:

```
(let*
  ((ResultPort (*process-receive 'pwd))
   (Result (read ResultPort)))
  (close ResultPort)
  Result)
```

### 18.7.3 The perils of pipes

In addition to the usual care which must be exercised by users of pipes, LISP programmers must observe additional cautions unique to FRANZ LISP. Pipes belong to the LISP process, not the executing LISP program.

Most important, each pipe uses a port from LISP's limited repertoire (on some operating systems, this limit can be as small as 20—on others there may be no limit). If a LISP



function does not close its pipes for any reason (e.g. the function dies), the ports continue to be *used* and the associated subprocesses continue to exist. If all the ports are used up while you are debugging a program or running an erroneous program that fails to close ports, you will get an error from the process-creation function that it cannot open a port.

Execution of LISP's **resetio** function which closes the open ports (with the exception of standard input and output) will cure this problem; however, this indiscriminately severs the connections to all subprocesses and kills them. Unless you run out of ports by erroneously leaving them in use, you should probably avoid the **resetio** function. (If you must call **resetio** to close all pipes, remember to call **wait** repeatedly; see below).

You should call **wait** every time a process dies. Calling **wait** ensures that a dying process does not linger as a zombie, which can cause you to exceed your process limit. (If a process creation function fails for no apparent reason, check for *zombies* by stopping the LISP process and running the Unix **ps** command.) **wait** returns a dotted-pair consisting of the terminated process's id and status. If no child processes of the current process exist or if an error occurs (e.g. an interrupted system call), **wait** returns with the process id set to -1. The **wait** function otherwise waits for a child process to terminate or to stop (4.2BSD Unix), or for a signal to be received.

## 19 Objects, message-passing, and flavors

- 19.1 Introduction 19-1
- 19.2 Objects 19-1
- 19.3 Modularity 19-3
- 19.4 Generic operations 19-6
- 19.5 Generic operations in LISP 19-7
- 19.6 Simple use of flavors 19-9
- 19.7 Mixing flavors 19-12
- 19.8 Flavor functions 19-16
- 19.9 Defflavor options 19-24
- 19.10 Flavor families 19-32
- 19.11 Vanilla flavor 19-32
- 19.12 Method combination 19-34
- 19.13 Implementation of flavors 19-42
  - 19.13.1 Order of definition 19-43
  - 19.13.2 Changing a flavor 19-44
- 19.14 Property list operations 19-45
- 19.15 Copying instances 19-46





## 19 Objects, message-passing, and flavors

### 19.1 Introduction

The object-oriented programming style used in the Smalltalk and Actor families of languages is available in FRANZ LISP. Its purpose is to perform *generic operations* on objects. Part of its implementation is simply a convention in procedure-calling style; part is a powerful language feature, called Flavors, for defining abstract objects. This chapter explains the principles of object-oriented programming and message passing, and the use of Flavors in implementing these in FRANZ LISP. It assumes no prior knowledge of any other languages.

There are two entirely separate implementations of Flavors distributed by Franz Inc. The earlier version is an adaptation, done at MIT, of an early MIT LISP Machine implementation. Starting with FRANZ LISP Opus 42, an entirely new, proprietary, and much more efficient implementation replaces the earlier version. (However, the earlier version is still distributed by Franz Inc. for backward compatibility.) This chapter documents only the newer proprietary version. The text of the chapter is a heavily-edited version of Chapter 20 from the MIT LISP Machine Manual, as made available through MIT's Project Athena. It has been subsequently edited by the staff of Franz Inc. for inclusion in the FRANZ LISP manual.

In general, the FRANZ LISP implementation of Flavors is quite similar to that in Zetalisp, although a few details and extensions differ. Most code should port easily between versions.

### 19.2 Objects

When writing a program, it is often convenient to model what the program does in terms of *objects*, conceptual entities that can be likened to real-world things. Choosing what objects to provide in a program is very important to the proper organization of the program. In an object-oriented design, specifying what objects exist is the first task in designing the system. In a text editor, the objects might be *pieces of text*, *pointers into text*, and *display windows*. In an electrical design system, the objects might be *resistors*, *capacitors*, *transistors*, *wires*, and *display windows*. After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object. In the text editor example, operations on *pieces of text* might include inserting text and deleting text; operations on *pointers into text* might include moving forward and



backward; and operations on *display windows* might include redisplaying the window and changing which *piece of text* the window is associated with.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it. More rigorously, the program defines several *types* of object (the editor above has three types), and it can create many *instances* of each type (that is, there can be many pieces of text, many pointers into text, and many windows). The program defines a set of types of object and, for each type, a set of operations that can be performed on any object of the type.

The new type abstractions may exist only in the programmer's mind. The mapping into a concrete representation may be done without the aid of any programming features. For example, it is possible to think of an atom's property list as an implementation of an abstract data type on which certain operations are defined, implemented in terms of the LISP functions **get** and **putprop**. As another example, it is also possible to use *disembodied property lists*. These are property lists (association lists of pairs) which are, however, not stored in the global structure of an atom. A disembodied property list has an odd number of items, the first being *nil*. **get** and **putprop** still work. This type can be instantiated with `(cons nil nil)` (that is, by evaluating this form you can create a new disembodied property list); the operations are invoked through functions defined just for that purpose. The fact that disembodied property lists are really implemented as lists, indistinguishable from any other lists, does not invalidate this point of view. However, such conceptual data types cannot be distinguished automatically by the system; one cannot ask: *is this object a disembodied property list, as opposed to an ordinary list?*

The **defstruct** for *ship* used as an example elsewhere in this manual is another application. This is reviewed in the next section. **defstruct** automatically defines some operations on the objects: the operations to access its elements. We could define other functions that did useful computation with ships, such as computing their speed, angle of travel, momentum, or velocity, stopping them, moving them elsewhere, and so on.

In both cases, we represent our conceptual object by one LISP object. The LISP object we use for the representation has *structure* and refers to other LISP objects. In the disembodied property list case, the LISP object is a list of pairs; in the ship case, the LISP object is an array or vector whose details are taken care of by **defstruct**. In both cases, we can say that the object keeps track of an *internal state*, which can be *examined* and *altered* by the operations available for that type of object. **get** examines the state of a property list, and **putprop** alters it; **ship-x-position** examines the state of a ship, and `(setf (ship-x-position ship) 5.0)` alters it.

This is the essence of object-oriented programming. A conceptual object is modeled by a single LISP object, which bundles up some state information. For every type of object, there is a set of operations that can be performed to examine or alter the state of the object.



### 19.3 Modularity

An important benefit of the object-oriented style is that it lends itself to a particularly simple and lucid kind of modularity. If you have modular programming constructs and techniques available, they help and encourage you to write programs that are easy to read and understand, and so are more reliable and maintainable. Object-oriented programming lets a programmer implement a useful facility that presents the caller with a set of external interfaces, without requiring the caller to understand how the internal details of the implementation work. In other words, a program that calls this facility can treat the facility as a black box; the calling program has an implicit contract with the facility guaranteeing the external interfaces, and that is all it knows.

For example, a program that uses disembodied property lists never needs to know that the property list is being maintained as a list of alternating indicators and values; the program simply performs the operations, passing them inputs and getting back outputs. The program only depends on the external definition of these operations: it knows that if it **putprops** a property, and doesn't **remprop** it (or **putprop** over it), then it can use **get** to be sure of getting back the same thing which was put in. This hiding of the details of the implementation means that someone reading a program that uses disembodied property lists need not concern himself with how they are implemented; he need only understand how what abstract operations are represented. This lets the programmer concentrate his energies on building a higher-level program rather than understanding the implementation of the support programs. This hiding of implementation means that the representation of property lists could be changed and the higher-level program would continue to work. For example, instead of a list of alternating elements, the property list could be implemented as an association list or a hash table. Nothing in the calling program would change at all.

The same is true of the ship example. The caller is presented with a collection of operations, such as **ship-x-position**, **ship-y-position**, **ship-speed**, and **ship-direction**; it simply calls these and looks at their answers, without caring how they did what they did. In our example above, **ship-x-position** and **ship-y-position** would be accessor functions, defined automatically by **defstruct**, while **ship-speed** and **ship-direction** would be functions defined by the implementor of the ship type. The code might look like this:

---

```
(defstruct (ship :conc-name)
  x-position
  y-position
  x-velocity
  y-velocity
  mass)

(defun ship-speed (ship)
  (sqrt (+ ( ^ (ship-x-velocity ship) 2)
           ( ^ (ship-y-velocity ship) 2)))))
```



```
(defun ship-direction (ship)
  (atan (ship-y-velocity ship)
        (ship-x-velocity ship)))
```

---

The caller need not know that the first two functions were structure accessors and that the second two were written by hand and perform arithmetic. Those facts would not be considered part of the black-box characteristics of the implementation of the ship type. The ship type does not guarantee which functions will be implemented in which ways; such aspects are not part of the contract between ship and its callers. In fact, ship could have been written this way instead:

---

```
(defstruct (ship :conc-name)
  x-position
  y-position
  speed
  direction
  mass)

(defun ship-x-velocity (ship)
  (* (ship-speed ship) (cos (ship-direction ship))))

(defun ship-y-velocity (ship)
  (* (ship-speed ship) (sin (ship-direction ship))))
```

---

In this second implementation of the ship type, we have decided to store the velocity in polar coordinates instead of rectangular coordinates. This is purely an implementation decision. The caller has no idea which of the two ways the implementation uses; he just performs the operations on the object by calling the appropriate functions.

We have now created our own types of objects, whose implementations are hidden from the programs that use them. Such types are usually referred to as *abstract types*. The object-oriented style of programming can be used to create abstract types by hiding the implementation of the operations and simply documenting what the operations are defined to do.

Some more terminology: the quantities being held by the elements of the ship structure are referred to as *instance variables*. Each instance of a type has the same operations defined on it; what distinguishes one instance from another (besides *eqness*) is the values that reside in its instance variables. The example above illustrates that a caller of operations does not know what the instance variables are; our two ways of writing the ship operations have different instance variables, but from the outside they have exactly the same operations.

One might ask: *but what if the caller evaluates (vref ship 2) and notices that he gets back the x-velocity rather than the speed? Then he can tell which of the two implementations were used.* This is true; if the caller were to do that, he could tell. However, when a facility is implemented in the object-oriented style, only certain functions are



documented and advertised, the functions that are considered to be operations on the type of object. The contract from ship to its callers only speaks about what happens if the caller calls these functions. The contract makes no guarantees at all about what would happen if the caller were to start poking around on his own using `vref`. A caller who does so is in error. He is depending on the concrete implementation of the abstraction: something that is not specified in the contract. No guarantees were ever made about the results of such action, and so anything may happen; indeed, if ship were reimplemented, the code that does the `vref` might have a different effect entirely and probably stop working. This example shows why the concept of a contract between a callee and a caller is important: the contract specifies the interface between the two modules.

Unlike some other languages that provide abstract types, FRANZ LISP makes no attempt to have the language automatically forbid constructs that circumvent the contract. This is intentional. One reason for this is that LISP is an interactive system, and so it is important to be able to examine and alter internal state interactively (usually from a debugger). Furthermore, there is no strong distinction between the *system* and the *user* portions of the FRANZ LISP system; users are allowed to get into nearly any part of the language system and change what they want to change.

In summary: by defining a set of operations and making only a specific set of external entry-points available to the caller, the programmer can create his own abstract types. These types can be useful facilities for other programs and programmers. Since the implementation of the type is hidden from the callers, modularity is maintained and the implementation can be changed easily.

We have hidden the implementation of an abstract type by making its operations into functions which the user may call. The importance of the concept is not that they are functions — in LISP everything is done with functions. The important point is that we have defined a new conceptual operation and given it a name, rather than requiring each user who wants to do the operation to write it out step-by-step. Thus we say (ship-x-velocity s) rather than (arrayref s 2).

Often a few abstract operation functions are simple enough that it is desirable to compile special code for them rather than really calling the function. (Compiling special code like this is often called *open-coding*.) The compiler is directed to do this through use of macros for example. `defstruct` arranges for this kind of special compilation for the functions that get the instance variables of a structure.

When we use this optimization, the implementation of the abstract type is only hidden in a certain sense. It does not appear in the LISP code written by the user, but does appear in the compiled code. The reason is that there may be some compiled functions that use the macros (or other concrete manifestation of the implementation). Even if you change the definition of the macro, the existing compiled code will continue to use the old definition. Thus, if the implementation of a module is changed, programs that use it may need to be recompiled. This sacrifice of compatibility between interpreted and compiled code is usually quite acceptable for the sake of efficiency in debugged code.

In the FRANZ LISP implementation of Flavors which is discussed below, there is no such compiler incorporation of nonmodular knowledge into a program, except when the *:ordered-instance-variables* feature is used (described below). If you don't use the



*:ordered-instance-variables* feature, you don't have to worry about this.

## 19.4 Generic operations

Consider the rest of the program that uses the ship abstraction. It may want to deal with other objects that are like ships in that they are movable objects with mass, but unlike ships in other ways. A more advanced model of a ship might include the concept of the ship's engine power, the number of passengers on board, and its name. An object representing a meteor probably would not have any of these, but might have another attribute such as how much iron is in it.

However, all kinds of movable objects have positions, velocities, and masses, and the system will contain some programs that deal with these quantities in a uniform way, regardless of what kind of object is being modeled. For example, a piece of the system that calculates every object's orbit in space need not worry about the other, more peripheral attributes of various types of objects; it works the same way for all objects. Unfortunately, a program that tries to calculate the orbit of a ship needs to know the ship's attributes, and must therefore call **ship-x-position** and **ship-y-velocity** and so on. The problem is that these functions won't work for meteors. There would have to be a second program to calculate orbits for meteors that would be exactly the same, except that where the first one calls **ship-x-position**, the second one would call **meteor-x-position**, and so on. This would be very bad; a great deal of code would have to exist in multiple copies, all of it would have to be maintained in parallel, and it would take up space for no good reason.

What is needed is an operation that can be performed on objects of several different types. For each type, it should do the thing appropriate for that type. Such operations are called *generic* operations. The classic example of generic operations is the arithmetic functions in many programming languages, including FRANZ LISP. The **plus** function accepts integers, floats or bignums and performs an appropriate kind of addition based on the data types of the objects being manipulated. In Macsyma, an algebraic manipulation system which runs under FRANZ LISP, the **+** operation works for matrices, polynomials, rational functions, and arbitrary algebraic expression trees. In our example, we need a generic **x-position** operation that can be performed on either ships, meteors, or any other kind of mobile object represented in the system. This way, we can write a single program to calculate orbits. When it wants to know the x position of the object it is dealing with, it simply invokes the generic **x-position** operation on the object, and whatever type of object it has, the correct operation is performed, and the x position is returned.

In the following discussion we use another idiom adopted from the Smalltalk language: performing a generic operation is called *sending a message*. The message consists of an operation name (a symbol) and arguments. The objects in the program are thought of as little people, who get sent messages and respond with answers (returned values). In the example above, the objects are sent **x-position** messages, to which they respond with their x position.

Sending a message is a way of invoking a function without specifying which function is to be called. Instead, the data determines the function to use. The caller specifies an



operation name and an object; that is, it said what operation to perform, and what object to perform it on. The function to invoke is found from this information.

The two data used to figure out which function to call are the *type* of the object, and the *name* of the operation. The same set of functions are used for all instances of a given type, so the type is the only attribute of the object used to figure out which function to call. The rest of the message besides the operation is data which are passed as arguments to the function, so the operation is the only part of the message used to find the function. Such a function is called a *method*. For example, if we send an **x-position** message to an object of type ship, then the function we find is *the ship type's x-position method*. A method is a function that handles a specific operation on a specific kind of object; this method handles messages named **x-position** to objects of type ship.

In our new terminology: the orbit-calculating program finds the x position of the object it is working on by sending that object a message consisting of the operation **x-position** and no arguments. The returned value of the message is the x position of the object. If the object was of type ship, then the ship type's **x-position** method was invoked; if it was of type meteor, then the meteor type's **x-position** method was invoked. The orbit-calculating program just sends the message, and the right function is invoked based on the type of the object. We now have true generic functions, in the form of message passing: the same operation can mean different things depending on the type of the object.

## 19.5 Generic operations in LISP

How do we implement message passing in LISP? Our convention is that objects that receive messages are always *functional* objects (that is, you can apply them to arguments). A message is sent to an object by calling that object as a function, passing the operation name as the first argument and the arguments of the message as the rest of the arguments. Operation names are represented by symbols; normally these symbols are in the keyword package, since messages may normally be passed between objects defined in different packages. So if we have a variable *my-ship* whose value is an object of type ship, and we want to know its x position, we send it a message as follows:

---

```
(send my-ship :x-position)
```

---

To set the ship's x position to 3.0, we send it a message like this:

---

```
(send my-ship :set-x-position 3.0)
```

---

A variation supported in some Flavor systems would allow



---

```
(send my-ship :set :x-position 3.0) ;; not supported
```

---

but this is now deprecated and not provided in FRANZ LISP.

It should be stressed that no new features are added to LISP for message sending; we simply define a convention on the way objects take arguments. The convention says that an object accepts messages by always interpreting its first argument as an operation name. The object must consider this operation name, find the function which is the method for that operation, and invoke that function.

To emphasize the relationship between well-known features and the new object-oriented version, we define the two basic functions for message passing as follows:

**(send 's\_object 's\_message ['argument ...])**

- This function is equivalent to **funcall**; however, the implementation may gain efficiency by assuming the **s\_object** is a flavor instance.
- Conceptually, this sends **s\_object** a message with operation and arguments as specified. The function **send** is preferable to **funcall** when a message is being sent, since it reminds the programmer of the usage.
- In some implementations of Flavors, the semantics of **send** may differ from **funcall** in those cases where **s\_object** is a symbol, list, number, or other object that does not normally handle messages.

**(lexpr-send 's\_object 's\_message ['argument ...] 'list-of-arguments)**

- This function is equivalent to **apply**, however, the implementation may gain efficiency by assuming the **s\_object** is a flavor instance. The last argument should be a list.

How does this all work? The object must somehow find the right method for the message it is sent. Furthermore, the object now has to be callable as a function. However, an ordinary function will not do: we need a data structure that can store the instance variables (the internal state) of the object. Of the FRANZ LISP features available, the most appropriate is the closure. A message-receiving object could be implemented as a closure over a set of instance variables. The function inside the closure would have a big **caseq** form to dispatch on its first argument.

While using closures as given does work, it has several problems. The main problem is that in order to add a new operation to a system, it is necessary to modify code in more than one place: you have to find all the types that *understand* that operation, and add a new clause to the **caseq**. The problem with this is that you cannot textually separate the implementation of your new operation from the rest of the system: the methods must be interleaved with the other operations for the type. Adding a new operation should only require *adding* LISP code; it should not require *modifying* LISP code.



For example, the conventional way of making generic operations for arithmetic on various new mathematical objects is to have a procedure for each operation (+, \*, etc), which has a big **caseq** for all the types; this means you have to modify code in **generic-plus**, **generic-times**, ... to add a type. This is inconvenient and error-prone.

The *flavor* mechanism is a streamlined, more convenient, and time-tested system for creating message-receiving objects. With flavors, you can add a new method simply by adding code, without modifying existing code. Furthermore, many common and useful things are very easy to do with flavors. The rest of this chapter describes flavors.

## 19.6 Simple use of flavors

A *flavor*, in its simplest form, is a definition of an abstract type. New flavors are created with the **defflavor** special form, and methods of the flavor are created with the **defmethod** special form. New instances of a flavor are created with the **make-instance** function. This section explains simple uses of these forms.

For an example of a simple use of flavors, here is how the ship example above would be implemented.

---

```
(defflavor ship (x-position y-position x-velocity y-velocity mass)
  ()
  :gettable-instance-variables)

(defmethod (ship :speed) ()
  (sqrt (+ (^ x-velocity 2)
            (^ y-velocity 2))))

(defmethod (ship :direction) ()
  (atan y-velocity x-velocity))
```

---

The code above creates a new flavor. The first subform of the **defflavor** is **ship**, which is the name of the new flavor. Next is the list of instance variables; they are the five that should be familiar by now. The next subform is something we will get to later. The rest of the subforms are the body of the **defflavor**, and each one specifies an option about this flavor. In our example, there is only one option, namely **:gettable-instance-variables**. This means that for each instance variable, a method should automatically be generated to return the value of that instance variable. The name of the operation is a symbol with the same name as the instance variable, but **interned** in the keyword package. Thus, methods are created to handle the operations **:x-position**, **:y-position**, and so on.

Each of the two **defmethod** forms adds a method to the flavor. The first one adds a handler to the flavor **ship** for the operation **:speed**. The second subform is the lambda-list, and the rest is the body of the function that handles the **:speed** operation. The body can refer to or set any instance variables of the flavor, just like variables bound by a



containing **let**. When any instance of the ship flavor is invoked with a first argument of **:direction**, the body of the second **defmethod** is evaluated in an environment in which the instance variables of ship refer to the instance variables of this instance (the one to which the message was sent). So the arguments passed to **atan** are the the velocity components of this particular ship. The result of **atan** becomes the value returned by the **:direction** operation.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor has the five instance variables named in the **defflavor** form, and the seven methods we have seen (five that were automatically generated because of the **:gettable-instance-variables** option, and two that we wrote ourselves). The way to create an instance of our new flavor is with the **make-instance** function. Here is how it could be used:

---

```
(setq my-ship (make-instance 'ship))
```

---

This returns an object whose printed representation is something like **#<ship 13731210>**. (the details of the print form will vary; it is an object which cannot be read back in from the short-hand printed representation). The argument to **make-instance** is the name of the flavor to be instantiated. Additional arguments, not used here, are *init options*, that is, commands to the flavor of which we are making an instance, selecting optional features. This will be discussed more in a moment.

Examination of the flavor we have defined shows that it is quite useless as it stands, since there is no way to set any of the parameters. We can fix this up easily by putting the **:settable-instance-variables** option into the **defflavor** form. This option tells **deflavor** to generate methods for operations **:set-x-position**, **:set-y-position**, and so on. Each such method takes one argument and sets the corresponding instance variable to that value.

Another option we can add to the **defflavor** is **:inittable-instance-variables**, (alternative spelling for compatibility is **:initable-instance-variables**) which allows us to initialize the values of the instance variables when an instance is first created. **:inittable-instance-variables** does not create any methods; instead, it makes *initialization keywords* named **:x-position**, **:y-position**, etc., that can be used as init-option arguments to **make-instance** to initialize the corresponding instance variables. The list of init options is sometimes called the *init-plist* because it is like a property list.

Here is the improved **defflavor**:

---

```
(defflavor ship (x-position y-position x-velocity y-velocity mass)
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)
```

---



All we have to do is evaluate this new **defflavor**, and the existing flavor definition is updated and now includes the new methods and initialization options. In fact, the instance we generated a while ago now accepts the new operations! We can set the mass of the ship we created by evaluating:

---

```
(send my-ship :set-mass 3.0)
```

---

and the mass instance variable of my-ship is properly set to 3.0.

If you want to play around with flavors, it is useful to know that **describe** of an instance tells you the flavor of the instance and the values of its instance variables. If we were to evaluate (**describe my-ship**) at this point, the following would be printed:

---

```
#<ship 3214320>, an object of flavor ship,
has instance variable values:
  x-position:      nil
  y-position:      nil
  x-velocity:      nil
  y-velocity:      nil
  mass:            3.0
```

---

Now that the instance variables are *inittable*, we can create another ship and initialize some of the instance variables using the **init-plist**. Let's do that and **describe** the result:

---

```
=> (setq her-ship (make-instance 'ship :x-position 0.0 :y-position 2.0
:mass 3.5))
```

```
#<ship 3242340>
```

```
=> (describe her-ship)
#<ship 3242340>, an object of flavor ship,
has instance variable values:
  x-position:      0.0
  y-position:      2.0
  x-velocity:      nil
  y-velocity:      nil
  mass:            3.5
```

---

A flavor can also establish default initial values for instance variables. These default values are used when a new instance is created if the values are not initialized any other way. The syntax for specifying a default initial value is to replace the name of the instance variable by a list, whose first element is the name and whose second is a form to evaluate to produce the default initial value. For example when read in the definitions:



---

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
  ()
  :gettable-instance-variables
  :settable-instance-variables
  :inittable-instance-variables)
```

---

Then the results are as follows:

---

```
=> (setq another-ship (make-instance 'ship :x-position 3.4))
#<ship 2342340>
=> (describe another-ship)
#<ship 2342340>
an object of flavor ship,
has instance variable values:
  x-position:      3.4
  y-position:      0.0
  x-velocity:      2.0
  y-velocity:      3.0
  mass:            nil
```

---

The value of `x-position` was initialized explicitly, so the default was ignored. The value of `y-position` was initialized from the default value, which was `0.0`. The two velocity instance variables were initialized from their default values, which came from two global variables. The value of `mass` was not explicitly initialized and did not have a default initialization, so it was left as `nil`. Some flavor implementations set an uninitialized instance variable to **unbound** rather than `nil`.

There are many other options that can be used in **defflavor**, and the `init` options can be used more flexibly than just to initialize instance variables; full details are given later in this chapter. But even with the small set of features we have seen so far, it is easy to write object-oriented programs.

## 19.7 Mixing flavors

Now we have a system for defining message-receiving objects so that we can have generic operations. If we want to create a new type called `meteor` that would accept the same generic operations as `ship`, we could simply write another **defflavor** and two more



**defmethods** that looked just like those of **ship**, and then **meteors** and **ships** would both accept the same operations. Objects of type **ship** would have some more instance variables for holding attributes specific to ships and some more methods for operations that are not generic, but are only defined for ships; the same would be true of **meteor**.

However, this would be a wasteful thing to do. The same code has to be repeated in several places, and several instance variables have to be repeated. The code now needs to be maintained in many places, which is always undesirable. The power of flavors (and the name *flavors*) comes from the ability to mix several flavors and get a new flavor. Since the functionality of **ship** and **meteor** partially overlap, we can take the common functionality and move it into its own flavor, which might be called **moving-object**. We would define **moving-object** the same way as we defined **ship** in the previous section. Then, **ship** and **meteor** could be defined like this:

---

```
(defflavor ship (engine-power number-of-passengers name)
               (moving-object)
               :gettable-instance-variables)

(defflavor meteor (percent-iron)
                 (moving-object)
                 :inittable-instance-variables)
```

---

These **defflavor** forms use the second subform, for which we previously used (). The second subform is a list of flavors to be combined to form the new flavor; such flavors are called *components*. Concentrating on **ship** for a moment (analogous statements are true of **meteor**), we see that it has exactly one component flavor: **moving-object**. It also has a list of instance variables, which includes only the ship-specific instance variables and not the ones that it shares with **meteor**. By incorporating **moving-object**, the **ship** flavor acquires all of its instance variables, and so need not name them again. It also acquires all of **moving-object**'s methods, too. So with the new definition, **ship** instances still implement the **:x-velocity** and **:speed** operations, with the same meaning as before. However, the **:engine-power** operation is also understood (and returns the value of the engine-power instance variable).

What we have done here is to take an abstract type, **moving-object**, and build two more specialized and powerful abstract types on top of it. Any **ship** or **meteor** can do anything a **moving-object** can do, and each also has its own specific abilities. This kind of building can continue; we could define a flavor called **ship-with-passenger** that was built on top of **ship**, and it would inherit all of **moving-object**'s instance variables and methods as well as **ship**'s instance variables and methods. Furthermore, the second subform of **defflavor** can be a list of several components, meaning that the new flavor should combine all the instance variables and methods of all the flavors in the list, as well as the ones *those* flavors are built on, and so on. All the components taken together form a big tree of flavors. A flavor is built from its components, its components' components, and so on. We sometimes use the term *components* to mean the immediate



components (the ones listed in the **defflavor**), and sometimes to mean all the components (including the components of the immediate components and so on). (Actually, it is not strictly a tree, since some flavors might be components through more than one path. It is really a directed graph; it can even be cyclic.)

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including non-terminal nodes *before* the subtrees they head, ignoring any flavor that has been encountered previously somewhere else in the tree. For example, if flavor-1's immediate components are flavor-2 and flavor-3, and flavor-2's components are flavor-4 and flavor-5, and flavor-3's component was flavor-4, then the complete list of components of flavor-1 would be: (*flavor-1, flavor-2, flavor-4, flavor-5, flavor-3*). The flavors earlier in this list are the more specific, less basic ones; in our example, ship-with-passengers would be first in the list, followed by ship, followed by moving-object. A flavor is always the first in the list of its own components. Notice that flavor-4 does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is done during the walk; a cycle in the directed graph does not cause a non-terminating computation.)

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both flavor-2 and flavor-3 have instance variables named *foo*, then flavor-1 has an instance variable named *foo*, and all methods that refer to *foo* refer to this same instance variable. Thus different components of a flavor can communicate with one another using shared instance variables. (Often, only one component ever sets the variable; the others only look at it.) The default initial value for an instance variable comes from the first component flavor to specify one.

The way the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each operation supported by the flavor. This function is constructed out of all the methods for that operation from all the components of the flavor. There are many different ways that methods can be combined; these can be selected by the user when a flavor is defined. The user can also create new forms of combination.

There are several kinds of methods, but so far, the only kinds of methods we have seen are *primary* methods. The default way primary methods are combined is that all but the earliest one provided are ignored. In other words, the combined method is simply the primary method of the first flavor to provide a primary method. What this means is that if you are starting with a flavor *foo* and building a flavor *bar* on top of it, then you can override *foo*'s method for an operation by providing your own method. Your method will be called, and *foo*'s will never be called.

Simple overriding is often useful; for example, if you want to make a new flavor *bar* that is just like *foo* except that it reacts completely differently to a few operations. However, often you don't want to completely override the base flavor's (*foo*'s) method; sometimes you want to add some extra things to be done. This is where combination of methods is used.

The usual way methods are combined is that one flavor provides a primary method, and other flavors provide *daemon methods*. The idea is that the primary method is in



*charge* of the main business of handling the operation, but other flavors just want to keep informed that the message was sent, or just want to do the part of the operation associated with their own area of responsibility.

*daemon* methods come in two kinds, *before* and *after*. There is a special syntax in **defmethod** for defining such methods. Here is an example of the syntax. To give the ship flavor an after-daemon method for the **:speed** operation, the following syntax would be used:

---

```
(defmethod (ship :after :speed) ( body))
```

---

Now, when a message is sent, it is handled by a new function called the *combined* method. The combined method first calls all of the before daemons, then the primary method, then all the after daemons. Each method is passed the same arguments that the combined method was given. The returned values from the combined method are the values returned by the primary method; any values returned from the daemons are ignored. Before-daemons are called in the order that flavors are combined, while after-daemons are called in the reverse order. In other words, if you build bar on top of foo, then bars before-daemons run before any of those in foo, and bars after-daemons run after any of those in foo.

The reason for this order is to keep the modularity order correct. If we create flavor-1 built on flavor-2, then the components of flavor-2 should not matter. Our new before-daemons go before all methods of flavor-2, and our new after-daemons go after all methods of flavor-2. Note that if you have no daemons, this reduces to the form of combination described above. The most recently added component flavor is the highest level of abstraction; you build a higher-level object on top of a lower-level object by adding new components to the front. The syntax for defining daemon methods can be found in the description of **defmethod** below.

To make this a bit more clear, let's consider a simple example that is easy to play with; the **:print-self** method. The LISP printer (i.e. the **print** function) prints instances of flavors by sending them **:print-self** messages. The first argument to the **:print-self** operation is a port (we can ignore the others for now), and the receiver of the message is supposed to print its printed representation on the port. In the ship example above, the reason that instances of the ship flavor printed the way they did is because the ship flavor was actually built on top of a very basic flavor called vanilla-flavor; this component is provided automatically by **defflavor**. It was vanilla-flavors **:print-self** method that was doing the printing. Now, if we give ship its own primary method for the **:print-self** operation, then that method completely takes over the job of printing: vanilla-flavors method will not be called at all. However, if we give ship a before-daemon method for the **:print-self** operation, then it will get invoked before the vanilla-flavor method, and so whatever it prints will appear before what vanilla-flavor prints. So we can use before-daemons to add prefixes to a printed representation; similarly, after-daemons can add suffixes.



There are other ways to combine methods besides daemons, but this way is the most common. The more advanced ways of combining methods are explained in a later section. The details of vanilla-flavor and what it does for you are also explained later.

## 19.8 Flavor functions

We've been using the following special form informally:

(**defflavor** flavor-name ([var] ...) ([flav] ...) [option] ...))

- WHERE flavor-name is a symbol which serves to name this flavor.
- The vars are the names of the instance-variables containing the local state for this flavor. A list of two elements: the name of an instance-variable and a default initialization form is also acceptable; the initialization form is evaluated when an instance of the flavor is created if no other initial value for the variable is obtained. If no initialization is specified, the variable has value *nil*.
- The flavs are the names of the component flavors out of which this flavor is built. The features of those flavors are inherited as described previously.
- Each *option* may be either a keyword symbol or a list of a keyword symbol and arguments. The options to **defflavor** are described under *Defflavor Options*, below.
- SIDE EFFECT: The symbol flavor-name is given a flavor property which is the internal data-structure containing the details of the flavor.
- NOTE: In FRANZ LISP objects which are instances of flavors are implemented as vectors.

(**typep** 's\_item &optional 's\_flavor)

- RETURNS if s\_flavor is provided and non-*nil*, *t* if the symbol s\_item is an instance of s\_flavor, *nil* otherwise. If s\_flavor is missing or *nil*, the symbol which is the name of the flavor of the instance is returned.
- NOTE: The name of this function is in flux. Presently in FRANZ LISP **typep** does not implement the documented function. Instead, the function **instance-typep** may be used, or if the *lmhacks* module is loaded, the function **extended-typep**.

---

```
=> (instance-typep my-ship 'ship)
t
=> (instance-typep my-ship)
ship
```

---

\*all-flavor-names\*

- A special variable containing a list of the names of all flavors that have ever been defflavored.



(defmethod (flavor-name [ method-type ] operation) lambda-list [form] ...)

■ WHERE flavor-name is a symbol which is the name of the flavor which is to receive the method. operation is a keyword symbol which names the operation to be handled. method-type is a keyword symbol for the type of method; it is omitted when you are defining a primary method. For some method-types, additional information is expected. It comes after operation.

■ SIDE EFFECT: defmethod defines a method, that is, a function to handle a particular operation for instances of a particular flavor. The meaning of method-type depends on what style of method combination is declared for this operation. For instance, if **:daemon** combination (the default style) is in use, method types **:before** and **:after** are allowed. See section 19.11 for a complete description of the way methods are combined.

□ lambda-list describes the arguments and **&aux** variables of the function. The first argument to the method, which is the operation name itself, is automatically handled and so is not included in lambda-list. Note that methods may not have unevaluated arguments; that is, they must be functions, not macros or special forms. The forms are the function body; the value of the last form is returned when the method is applied. Some methods can return multiple values, depending on the style of method combination used.

□ The variant form

---

(defmethod (flavor-name operation) function)

---

where function is a symbol, says that flavor-names method for operation is function, a symbol which names a function. When function is called, self and any special instance variables will be bound. The function must take appropriate arguments; the first argument is the operation. Various flavor implementations have different conventions for automatically-supplied arguments to method functions; these should be conditionalized if code must be transportable.

If you redefine a method that is already defined, the new definition replaces the old one. Given a flavor, an operation name, and a method type, there can only be one function (with the exception of **:case** methods), so if you define a **:before** daemon method for the foo flavor to handle the **:bar** operation, then you replace the previous before-daemon; however, you do not affect the primary method or methods of any other type, operation or flavor.

Along with other things, defmethod causes a function to be defuned. The function name is formed by concatenating the hyphen-separated print names of all the symbols in the first defmethod subform, then suffixing **-method**; this name is interned in the same package as the flavor name. For example, (defmethod (foo :before :bar) ...) defines a function named **foo-before-bar-method**. This is useful to know if you want to trace a method, or if you want to poke around at the method function itself.



(**make-instance** 'flavor-name ['init-option 'value] ... )

- RETURNS an instance of the specified flavor which has just been created.
- Arguments after the first are alternating init-option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options, as described above. An **:init** message is sent to the newly-created object with one argument, the init-plist. This is a disembodied property-list containing the init-options specified and those defaulted from the flavor's **:default-init-plist** (however, init keywords that simply initialize instance variables, and the corresponding values, may be absent when the **:init** methods are called). **make-instance** is an easy-to-call interface to **instantiate-flavor**, below.

(**instantiate-flavor** 'flavor-name 'init-plist ['send-init-message-p 'return-unhandled-keywords 'area])

- RETURNS an instance.
- NOTE: This is an extended version of **make-instance**, giving you more features. Note that it takes the init-plist as an individual argument, rather than taking a **&rest** argument of init options and values.
- The init-plist argument must be a disembodied property list (beware!). This property list can be modified; the properties from the default init-plist are **putproped** on if not already present, and some **:init** methods do explicit **putprops** onto the init-plist.
- In the event that **:init** methods **remprop** properties already on the init-plist (as opposed to simply doing **get** and **putprop**), then the init-plist is **rplacded**. This means that the actual supplied list of options is modified. It also means that the caller of **instantiate-flavor** should copy its init-plist argument (e.g. with **append**).
- Do not use **nil** as the init-plist argument. This would mean to use the properties of the symbol **nil** as the init options. If your goal is to have no init options, you must provide a property list containing no properties, such as the list (**nil**).
- Here is the sequence of actions by which **instantiate-flavor** creates a new instance:
  - 1 The specified flavor's instantiation flavor function if it exists, is called to determine which flavor should actually be instantiated. If there is no instantiation flavor function, the specified flavor is instantiated.
  - If the flavor's method hash-table and other internal information have not been computed or are not up to date, they are computed. This may take a substantial amount of time, but it happens only once for each time you define or redefine a particular flavor.
  - 2 The instance itself is created. The area argument is ignored by FRANZ LISP and refers to consing in specified areas, a feature used in some LISP machines.
  - 3 Initial values of the instance variables are computed. If an instance variable is declared inittable, and a keyword with the same spelling as its name appears in init-plist, the property for that keyword is used as the initial value.
  - Otherwise, if the default init-plist specifies such a property, the value form is evaluated and the result used. Otherwise, if the flavor definition specifies a



default initialization form, it is evaluated and that result is used. In either case, the initialization forms may not refer to any instance variables, nor will they find the variable `self` be bound to the new instance. The value forms are evaluated before the instance is actually allocated.

- If an instance variable does not get initialized either of these ways it is left *nil*; an **`:init`** method may initialize it (see below).
- All remaining keywords and values specified in the **`:default-init-plist`** option to **`defflavor`**, that do not initialize instance variables and are not overridden by anything explicitly specified in `init-plist` are then merged into `init-plist` using **`putprop`**. The default `init` plist of the instantiated flavor is considered first, followed by those of all the component flavors in the standard order.
- 4 Keywords appearing in the `init-plist` but not defined with the **`:init-keywords`** option or the **`:inittable-instance-variables`** option for some component flavor are collected. If the **`:allow-other-keys`** option is specified with a non-*nil* value (either in the original `init-plist` argument or by some default `init` plist) then these *unhandled* keywords are ignored. If the `return-unhandled-keywords` argument is non-*nil*, a list of these keywords is returned as the second value of **`instantiate-flavor`**. Otherwise, an error is signaled if any unrecognized `init` keywords are present.
- If the `send-init-message-p` argument is supplied and non-*nil*, an **`:init`** message is sent to the newly-created instance, with one argument, the `init-plist`. **`get`** can be used to extract options from this property-list. Each flavor that needs initialization can contribute an **`:init`** method by defining a daemon.
- The **`:init`** methods should not look on the `init-plist` for keywords that simply initialize instance variables (that is, keywords defined with **`:inittable-instance-variables`** rather than **`:init-keywords`**). The corresponding instance variables are already set up when the `:init` methods are called, and sometimes the keywords and their values may actually be missing from the `init-plist` if it is more efficient not to put them on. To avoid problems, always refer to the instance variables themselves rather than looking for the `init` keywords that initialize them.

#### **`:init`** `init-plist`

- This operation is implemented on all flavor instances.
- SIDE EFFECT: This function examines the `init` keywords and perform whatever initializations are appropriate. `init-plist` is the argument that was given to **`instantiate-flavor`**, and may be passed directly to **`get`** to examine the value of any particular `init` option.
- The default definition of this operation does nothing. However, many flavors add **`:before`** and **`:after`** daemons to it.

#### **`(instancep 'object)`**

- RETURNS *t* if `object` is an instance of a flavor.



(**defwrapper** (flavor-name operation) lambda-list &body body)

■ NOTE: This feature is complex and you may not be able to understand it completely until you have gained some experience with flavors. It can safely be skipped meanwhile.

□ Sometimes the way the flavor system combines the methods of different flavors (the daemon system) is not powerful enough. In that case **defwrapper** can be used to define a macro that expands into code that is wrapped around the invocation of the methods. This is best explained by an example; suppose you needed a lock locked during the processing of the **:foo** operation on flavor bar, which takes two arguments, and you have a lock-frobboz special-form that knows how to lock the lock (presumably it generates an **unwind-protect**). lock-frobboz needs to see the first argument to the operation; perhaps that tells it what sort of operation is going to be performed (read or write).

---

```
(defwrapper (bar :foo) ((arg1 arg2) . body)
  `(lock-frobboz (self arg1)
    ' ,body))
```

---

The use of the **body** macro-argument prevents the macro defined by **defwrapper** from knowing the exact implementation and allows several **defwrappers** from different flavors to be combined properly.

Note that the argument variables, **arg1** and **arg2**, are not referenced with commas before them. These may look like **defmacro** argument variables, but they are not. Those variables are not bound at the time the **defwrapper**-defined macro is expanded and the back-quoting is done; rather, the result of that macro-expansion and back-quoting is code which, when a message is sent, will bind those variables to the arguments in the message as local variables of the combined method.

Consider another example. Suppose you thought you wanted a **:before** daemon, but found that if the argument was *nil* you needed to return from processing the message immediately, without executing the primary method. You could write a wrapper such as:

---

```
(defwrapper (bar :foo) ((arg1) . body)
  `(cond ((null arg1))
    (t (print "About to do :FOO")
      ' ,body)))
```

---

Suppose you need a variable for communication among the daemons for a particular operation; perhaps the **:after** daemons need to know what the primary method did, and it is something that cannot be easily deduced from just the arguments. You might use an instance variable for this, or you might create a special variable which is



bound during the processing of the operation and used free by the methods.

---

```
(defvar *communication*)
(defwrapper (bar :foo) (ignore . body)
  '(let ((*communication* nil))
    ' ,body))
```

---

Similarly you might want a wrapper that puts a **catch** around the processing of an operation so that any one of the methods could **throw** out in the event of an unexpected condition.

Like daemon methods, wrappers work in outside-in order; when you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component flavors. However, *all* wrappers happen before *any* daemons happen. When the combined method is built, the calls to the before-daemon methods, primary methods, and after-daemon methods are all placed together, and then the wrappers are wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new flavors execute within that wrapper's context.

Be careful about inserting the body into an internal lambda-expression within the wrapper's code. Doing so interacts with the internals of the flavor system and requires knowledge of things not documented in the manual in order to work properly.

**(defwhopper (flavor-name operation) lambda-list &body body)**

■ **NOTE:** Whoppers are a feature of some flavor implementations which can do many of the same things that wrappers can. They will be documented when they are implemented in FRANZ LISP.

**(undefmethod flavor [type] operation [suboperation])**

■ Removes a method: (undefmethod (flavor :before :operation)) removes the method created by (defmethod (flavor :before :operation) ...) To remove a wrapper, use **undefmethod** with **:wrapper** as the method type.

**(undefflavor 'flavor)**

■ Undefines flavor flavor. All methods of the flavor are lost. flavor and all flavors that depend on it are no longer valid to instantiate. If instances of the discarded definition exist, they continue to use that definition.

**self**

■ When a message is sent to an object, the special variable **self** is automatically bound to that object for the benefit of methods which want to manipulate the object itself (as opposed to its instance variables).



(send 's\_instance 's\_message ['argument ...])  
(funcall 's\_instance 's\_message ['argument ...])

■ NOTE: This is the way a message is passed to an instance of a flavor. **send** and **funcall** operate in essentially the same manner. **send** is potentially slightly more efficient because the evaluator can infer that the functional argument is an instance, whereas **funcall** must determine the type of its first argument.

(send-self 's\_message ['argument] ...)  
(funcall-self 's\_message ['argument] ...)  
(lexpr-send-self 's\_message ['argument ...] 'list-of-arguments)  
(lexpr-funcall-self 's\_message ['argument ...] 'list-of-arguments)

■ **funcall-self** is nearly equivalent to **funcall** with **self** as the first argument, but **funcall-self** is a little faster. The others are analogous.

(recompile-flavor 'flavor-name &optional 'single-op 'use-old-combined-methods 'do-dependents)

■ Updates the internal data of the flavor and any flavors that depend on it. If **single-op** is supplied non-*nil*, only the methods for that operation are changed. The system does this when you define a new method that did not previously exist. If **use-old-combined-methods** is *t*, then the existing combined method functions are used if possible. New ones are generated only if the set of methods to be called has changed. This is the default. If **use-old-combined-methods** is *nil*, automatically-generated functions to call multiple methods or to contain code generated by wrappers are regenerated unconditionally. If **do-dependents** is *nil*, only the specific flavor you specified is recompiled. Normally all flavors that depend on it are also recompiled.

□ **recompile-flavor** affects only flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mix-ins.

(compile-flavor-methods flavor-name ...)

■ The form (compile-flavor-methods flavor-name-1 flavor-name-2 ...), placed in a file to be compiled, directs the compiler to perform flavor combination for the named flavors, forcing the generation of the automatically-generated combined methods at compile time. This way, combined methods will run as compiled code instead of interpreted. Furthermore, the internal data structures needed to instantiate the flavor will be computed at load time, rather than waiting for the first attempt to instantiate the flavor.

□ You should only use **compile-flavor-methods** on a flavor that is going to be instantiated. For a flavor that is never going to be instantiated (that is, a flavor that only serves to be a component of other flavors that actually do get instantiated), it is a complete waste of time, except in the unusual case where those other flavors can



inherit the combined methods of this flavor instead of each one having its own copy of the combined method which happens to be identical to the others. In this unusual case, you should use the **:abstract-flavor** option to **defflavor**.

□ **compile-flavor-methods** forms should be compiled after all of the other information needed to create the combined methods is available. You should put them after all the definitions of all relevant flavors, wrappers, and methods of all components of the argument flavors.

□ When a **compile-flavor-methods** form is seen by the interpreter, the internal data structures are generated and the combined methods are defined.

■ **NOTE:** Some implementations invoke the compiler on combined methods dynamically from the interpreter whenever combined methods are generated; at present, this does not happen in FRANZ LISP.

(get-handler-for 'object 'operation)

■ Given an object and an operation, this returns the object's method for that operation, or *nil* if it has none. When object is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method.

□ This is related to the **:handler** function specification. It is preferable to use the generic operation **:get-handler-for**.

(flavor-allows-init-keyword-p 'flavor-name 'keyword)

■ **RETURNS** non-*nil* if the flavor named flavor-name allows keyword in the init options when it is instantiated, or *nil* if it does not. The non-*nil* value is the name of the component flavor that contributes the support of that keyword.

(si:flavor-allowed-init-keywords 'flavor-name)

■ **RETURNS** a list of all the init keywords that may be used in instantiating flavor-name.

(symeval-in-instance 'instance 'symbol [no-error-p])

■ **RETURNS** the value of the instance variable symbol inside instance. If there is no such instance variable, an error is signaled, unless no-error-p is non-*nil*, in which case *nil* is returned.

(set-in-instance 'instance 'symbol 'value)

■ **SIDE EFFECT:** Sets the value of the instance variable symbol inside instance to value. If there is no such instance variable, an error is signaled.

(describe-flavor 'flavor-name)

■ **SIDE EFFECT:** Prints descriptive information about a flavor; it is self-explanatory. An important thing it tells you that can be hard to figure out yourself is the combined



list of component flavors; this list is what is printed after the phrase 'and directly or indirectly depends on.'

## 19.9 Defflavor options

There are quite a few options to **defflavor**. They are all described here, although some are for very specialized purposes and not of interest to most users. A few options take additional arguments, and these are listed and described with the option.

Several of these options declare things about instance variables. These options can be given with arguments which are instance variables, or without any arguments in which case they refer to all of the instance variables listed at the top of the **defflavor**. This is *not* necessarily all the instance variables of the component flavors, just the ones mentioned in this flavor's **defflavor**. When arguments are given, they must be instance variables that were listed at the top of the **defflavor**; otherwise they are assumed to be misspelled and an error is signaled. It is legal to declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**.

### **:gettable-instance-variables**

- Enables automatic generation of methods for getting the values of instance variables. The operation name is the name of the variable, in the keyword package (i.e. it has a colon in front of it).
- Note that there is nothing special about these methods; you could easily define them yourself. This option generates them automatically to save you the trouble of writing out a lot of very simple method definitions. (The same is true of methods defined by the **:settable-instance-variables** option.) If you define a method for the same operation name as one of the automatically generated methods, the explicit definition overrides the automatic one.

### **:settable-instance-variables**

- Enables automatic generation of methods for setting the values of instance variables. The operation name is **:set-** followed by the name of the variable. All settable instance variables are also automatically made gettable and initable. (See the note in the description of the **:gettable-instance-variables** option, above.)

### **:initable-instance-variables**

- The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *initable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an init-option argument to **make-instance**. For compatibility with certain other implementations, the spelling **:inittable-instance-variables** is also accepted.



**:special-instance-variables**

■ The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, will be bound dynamically when handling messages. (By default, instance variables are bound lexically with the scope being the method.) You must do this to any instance variables that you wish to be accessible through **symeval** or **set**, since they see only dynamic bindings.

□ This should also be done for any instance variables that are declared globally special. If you omit this, the flavor system won't know about the special declaration, and will bind the instance variable lexically.

■ NOTE: In FRANZ LISP, instance variables properly default to lexical scope in compiled method code, but since the interpreter does not implement lexical scoping, all instance variables are bound as specials when an interpreted method function is entered.

□ Also, it is an error to apply **makunbound** to a special instance variable in FRANZ LISP since instances cannot store an unbound value.

**:init-keywords**

■ The arguments are declared to be valid keywords to use in **instantiate-flavor** when creating an instance of this flavor (or any flavor containing it). The system uses this for error-checking: before the system sends the **:init** message, it makes sure that all the keywords in the init-plist are either inittable instance variables or elements of this list. If any is not recognized, an error is signaled. When you write a **:init** method that accepts some keywords, they should be listed in the **:init-keywords** option of the flavor. If **:allow-other-keys** is used as an init keyword with a non-*nil* value, this error check is suppressed. Then unrecognized keywords are simply ignored.

**:default-init-plist**

■ The arguments are alternating keywords and value forms, like a property list. When the flavor is instantiated, these properties and values are put into the init-plist unless already present. This allows one component flavor to default an option to another component flavor. The value forms are only evaluated when and if they are used. For example,

---

```
(:default-init-plist :frob-array (make-array 100))
```

---

would provide a default *frob array* for any instance for which the user did not provide one explicitly. The following specification prevents errors for unhandled init keywords in all instantiations of this flavor and other flavors that depend on it.



---

```
(:default-init-plist :allow-other-keys t)
```

---

***:required-init-keywords*** *defflavor*

■ The arguments are init keywords which are to be required each time this flavor (or any flavor containing it) is instantiated. An error is signaled if any required init keyword is missing.

***:required-instance-variables***

■ Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables. An error occurs if there is an attempt to instantiate a flavor that incorporates this one if it does not have these in its set of instance variables. Note that this option is not one of those that checks the spelling of its arguments in the way described at the start of this section (if it did, it would be useless).

□ Required instance variables may be freely accessed by methods just like normal instance variables. The difference between listing instance variables here and listing them at the front of the **defflavor** is that the latter declares that this flavor *owns* those variables and accepts responsibility for initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

***:required-methods***

■ The arguments are names of operations that any flavor incorporating this one must handle. An error occurs if there is an attempt to instantiate such a flavor and it is lacking a method for one of these operations. Typically this option appears in the **defflavor** for a base flavor. Usually this is used when a base flavor does a (send self ...) to send itself a message that is not handled by the base flavor itself; the idea is that the base flavor will not be instantiated alone, but only with other components (mix-ins) that do handle the message. This keyword allows the error of having no handler for the message to be detected when the flavor instantiated or when **compile-flavor-methods** is done, rather than when the missing operation is used.

***:required-flavors***

■ The arguments are names of flavors that any flavor incorporating this one must include as components, directly or indirectly. The difference between declaring flavors as required and listing them directly as components at the top of the **defflavor** is that declaring flavors to be required does not make any commitments about where those flavors will appear in the ordered list of components; that is left up to whoever



does specify them as components. Declaring a flavor to be required only provides error checking: an attempt to instantiate a flavor that does not include the required flavors as components signals an error. Compare this with **:required-methods** and **:required-instance-variables**.

□ For an example of the use of required flavors, consider the ship example given earlier, and suppose we want to define a *relativity-mixin* which increases the mass dependent on the speed. We might write,

---

```
(defflavor relativity-mixin () (moving-object))
(defmethod (relativity-mixin :mass) ()
  (/ mass (sqrt (- 1
                  (^ (/ (send self :speed)
                       *speed-of-light*)
                     2)))))
```

---

but this would lose because any flavor that had *relativity-mixin* as a component would get *moving-object* right after it in its component list. As a base flavor, *moving-object* should be last in the list of components so that other components mixed in can replace its methods and so that daemon methods combine in the right order. *relativity-mixin* has no business changing the order in which flavors are combined, which should be under the control of its caller. For example,

---

```
(defflavor starship () (relativity-mixin long-distance-mixin ship))
```

---

puts *moving-object* last (inheriting it from *ship*). So instead of the definition above we write,

---

```
(defflavor relativity-mixin () () (:required-flavors moving-object))
```

---

which allows *relativity-mixins* methods to access *moving-object* instance variables such as *mass* (the rest mass), but does not specify any place for *moving-object* in the list of components.

□ It is very common to specify the *base flavor* of a mixin with the **:required-flavors** option in this way.

### **:included-flavors**

■ The arguments are names of flavors to be included in this flavor. The difference between declaring flavors here and declaring them at the top of the **defflavor** is that when component flavors are combined, if an included flavor is not specified as a



normal component, it is inserted into the list of components immediately after the last component to include it. Thus included flavors act like defaults. The important thing is that if an included flavor *is* specified as a component, its position in the list of components is completely controlled by that specification, independently of where the flavor that includes it appears in the list.

☐ ***:included-flavors*** and ***:required-flavors*** are used in similar ways; it would have been reasonable to use ***:included-flavors*** in the relativity-mixin example above. The difference is that when a flavor is required but not given as a normal component, an error is signaled, but when a flavor is included but not given as a normal component, it is automatically inserted into the list of components at a reasonable place.

#### ***:no-vanilla-flavor***

■ Normally when a flavor is instantiated, the special flavor ***si:vanilla-flavor*** is included automatically at the end of its list of components. The vanilla flavor provides some default methods for the standard operations which all objects are supposed to understand. These include ***:print-self***, ***:describe***, ***:which-operations***, and several other operations.

☐ If any component of a flavor specifies the ***:no-vanilla-flavor*** option, then ***si:vanilla-flavor*** is not included in that flavor. This option should not be used casually.

#### ***:default-handler***

■ The argument is the name of a function that is to be called to handle any operation for which there is no method. Its arguments are the arguments of the ***send*** which invoked the operation, including the operation name as the first argument. Whatever values the default handler returns are the values of the operation.

☐ Default handlers can be inherited from component flavors. If a flavor has no default handler, any operation for which there is no method causes an unclaimed message error.

#### ***:ordered-instance-variables***

■ This option is mostly for esoteric internal system uses. The arguments are names of instance variables which must appear first (and in this order) in all instances of this flavor, or any flavor depending on this flavor. This is used for instance variables that are specially known about by other code (e.g. non-LISP) and also in connection with the ***:outside-accessible-instance-variables*** option. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this ***defflavor***.

☐ Any number of flavors to be combined together can specify this option. The longest ordered variable list applies, and an error is signaled if any of the other lists do not match its initial elements.

☐ Removing any of the ***:ordered-instance-variables***, or changing their positions in the list, requires that you recompile all methods that use any of the affected instance variables.



**:outside-accessible-instance-variables**

■ The arguments are instance variables which are to be accessible from outside of this flavor's methods. A macro is defined (actually a **defsubst**, that is, both a **cmacro** and a function) which takes an object of this flavor as an argument and returns the value of the instance variable; **setf** may be used to set the value of the instance variable. The name of the macro is the name of the flavor concatenated with a hyphen and the name of the instance variable. These macros are similar to the accessor macros created by **defstruct**.

□ This feature works in two different ways, depending on whether or not the instance variable has been declared to have a fixed slot in all instances, via the **:ordered-instance-variables** option.

□ If the variable is not ordered, the position of its value cell in the instance must be computed at run time. This takes noticeable time, although less than actually sending a message would take. An error is signaled if the argument to the accessor macro is not an instance or is an instance that does not have an instance variable with the appropriate name. However, there is no error check that the flavor of the instance is the flavor the accessor macro was defined for, or a flavor built upon that flavor. This error check would be too expensive.

□ If the variable is ordered, the compiler compiles a call to the accessor macro into a primitive (actually a **vref**) which simply accesses that variable's assigned slot by number. No error-checking is performed to make sure that the argument is really an instance, much less that it is of the appropriate type.

□ **setf** works on these accessor macros to modify the instance variable.

**:accessor-prefix**

■ Normally the accessor macro created by the **:outside-accessible-instance-variables** option to access the flavor *fs* instance variable *v* is named *f-v*. This option allows something other than the flavor name to be used for the first part of the macro name. Specifying (**:accessor-prefix** *get\$*) causes it to be named *get\$v* instead.

**:alias-flavor**

■ **:alias-flavor** is presently unimplemented in FRANZ LISP.

□ Marks this flavor as being an alias for another flavor. This flavor should have only one component, which is the flavor it is an alias for, and no instance variables or other options. No methods should be defined for it.

□ The effect of the **:alias-flavor** option is that an attempt to instantiate this flavor actually produces an instance of the other flavor. Without this option, it would make an instance of this flavor, which might behave identically to an instance of the other flavor. **:alias-flavor** eliminates the need for separate mapping tables, method tables, etc. for this flavor, which becomes truly just another name for its component flavor.

□ The alias flavor and its base flavor are also equivalent when used as an argument of **subtypep** or as the second argument of **typep**; however, if the alias status of a flavor is changed, you must recompile any code which uses it as the second argument to



**typep** in order for such code to function.

☐ **:alias-flavor** is mainly useful for changing a flavor's name gracefully.

### **:abstract-flavor**

■ This option marks the flavor as one that is not supposed to be instantiated (that is, is supposed to be used only as a component of other flavors). An attempt to instantiate the flavor signals an error.

☐ It is sometimes useful to do **compile-flavor-methods** on a flavor that is not going to be instantiated, if the combined methods for this flavor will be inherited and shared by many others. **:abstract-flavor** tells **compile-flavor-methods** not to complain about missing required flavors, methods or instance variables. Presumably the flavors that depend on this one and actually are instantiated will supply what is lacking.

■ NOTE: **:abstract-flavor** is accepted but currently ignored in FRANZ LISP.

### **:method-combination**

■ Specifies the method combination style to be used for certain operations. Each argument to this option is a list (style order operation1 operation2...). operation1, operation2, etc. are names of operations whose methods are to be combined in the declared fashion. style is a keyword that specifies a style of combination. order is a keyword whose interpretation is up to style; typically it is either **:base-flavor-first** or **:base-flavor-last**.

☐ Any component of a flavor may specify the type of method combination to be used for a particular operation. If no component specifies a style of method combination, then the default style is used, namely **:daemon**. If more than one component of a flavor specifies the combination style for a given operation, then they must agree on the specification, or else an error is signaled.

### **:run-time-alternatives defflavor**

#### **:mixture defflavor**

■ A run-time-alternative flavor defines a collection of similar flavors, all built on the same base flavor but having various mixins as well. Instantiation chooses a flavor of the collection at run time based on the init keywords specified, using an automatically generated instantiation flavor function.

☐ A simple example would be

---

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives (:big big-foo-mixin))
  (:init-keywords :big))
```

---



- Then (make-instance 'foo :big t) makes an instance of a flavor whose components are big-foo-mixin as well as foo. But (make-instance 'foo) or (make-instance 'foo :big nil) makes an instance of foo itself. The clause (:big big-foo-mixin) in the **:run-time-alternatives** says to incorporate big-foo-mixin if **:big**s value is *t*, but not if it is *nil*.
- There may be several clauses in the **:run-time-alternatives**. Each one is processed independently. Thus, two keywords **:big** and **:wide** could independently control two mixins, giving four possibilities.

---

```
(deflavor foo () (basic-foo)
  (:run-time-alternatives (:big big-foo-mixin)
                          (:wide wide-foo-mixin))
  (:init-keywords :big))
```

---

- It is possible to test for values other than *t* and *nil*. The clause:

---

```
(:size (:big big-foo-mixin)
       (:small small-foo-mixin)
       (nil nil))
```

---

allows the value for the keyword **:size** to be **:big**, **:small** or *nil* (or omitted). If it is *nil* or omitted, no mixin is used (that's what the second *nil* means). If it is **:big** or **:small**, an appropriate mixin is used. This kind of clause is distinguished from the simpler kind by having a list as its second element. The values to check for can be anything, but **eq** is used to compare them.

- The value of one keyword can control the interpretation of others by nesting clauses within clauses. If an alternative has more than two elements, the additional elements are subclauses which are considered only if that alternative is selected. For example, the clause:

---

```
(:etherial (t etherial-mixin)
          (nil nil
            (:size (:big big-foo-mixin)
                  (:small small-foo-mixin)
                  (nil nil))))
```

---

says to consider the **:size** keyword only if **:etherial** is *nil*.

- **:mixture** is synonymous with **:run-time-alternatives**. It exists for compatibility with ZetaLISP or other LISP Machine systems.



**:documentation**

- Specifies the documentation string for the flavor definition, which is made accessible through (documentation flavorname 'flavor).
- This documentation can be viewed with the **describe-flavor** function.

## 19.10 Flavor families

The following organization conventions are recommended for programs that use flavors.

A *base flavor* is a flavor that defines a whole family of related flavors, all of which have that base flavor as a component. Typically the base flavor includes things relevant to the whole family, such as instance variables, **:required-methods** and **:required-instance-variables** declarations, default methods for certain operations, **:method-combination** declarations, and documentation on the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most cannot be instantiated themselves. They serve as a base upon which to build other flavors. The base flavor for the *foo* family is often named *basic-foo*.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated, because it is not a complete description. Each module or feature of a program is defined as a separate mixin; a usable flavor can be constructed by choosing the mixins for the desired characteristics and combining them, along with the appropriate base flavor. By organizing your flavors this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does, because the order of flavor combination controls the order in which daemons are invoked and wrappers are wrapped. Such order dependencies should be documented as part of the conventions of the appropriate family of flavors. A mixin flavor that provides the *mumble* feature is often named *mumble-mixin*.

If you are writing a program that uses someone else's facility, using that facility's flavors and methods, your program may still define its own flavors, in a simple way. The facility provides a base flavor and a set of mixins: the caller can combine these in various ways depending on exactly what it wants, since the facility probably does not provide all possible useful combinations. Even if your private flavor has exactly the same components as a pre-existing flavor, it can still be useful since you can use its **:default-init-plist** to select options of its component flavors and you can define one or two methods to customize it *just a little*.

## 19.11 Vanilla flavor

The operations described in this section are a standard protocol, which all message-receiving objects are assumed to understand. The standard methods that implement this protocol are automatically supplied by the flavor system unless the user specifically tells it not to do so. These methods are associated with the flavor *si:vanilla-flavor*:



**si:vanilla-flavor**

■ Unless you specify otherwise (with the **:no-vanilla-flavor** option to **defflavor**), every flavor includes the *vanilla* flavor, which has no instance variables but provides some basic useful methods.

**:print-self** stream prindepth escape-p

■ The object should output its printed-representation to a stream. The printer sends this message when it encounters an instance or an entity. The arguments are the stream, the current depth in list-structure (for comparison with **prinlevel**), and whether escaping is enabled (a copy of the value of *\*print-escape\**. **si:vanilla-flavor** ignores the last two arguments and prints something like *#<flavor-name octal-address>*. The *flavor-name* tells you what type of object it is and the *octal-address* allows you to tell different objects apart.

**:describe**

■ The object should describe itself, printing a description onto the standard output stream. The **describe** function sends this message when it encounters an instance. **si:vanilla-flavor** outputs in a reasonable format the object, the name of its flavor, and the names and values of its instance-variables. The instance variables are printed in their order within the instance.

**:which-operations**

■ The object should return a list of the operations it can handle. **si:vanilla-flavor** generates the list once per flavor and remembers it, minimizing consing and compute-time. If the set of operations handled is changed, this list is regenerated the next time someone asks for it.

**:operation-handled-p** operation

■ *operation* is an operation name. The object should return *t* if it has a handler for the specified operation, *nil* if it does not.

**:get-handler-for** operation

■ *operation* is an operation name. The object should return the method it uses to handle *operation*. If it has no handler for that operation, it should return *nil*. This is like the **get-handler-for** function.

**:send-if-handles** operation [arguments]...

■ *operation* is an operation name and *arguments* is a list of arguments for the operation. If the object handles the operation, it should send itself a message with that operation and arguments, and return whatever values that message returns. If it doesn't handle the operation it should just return *nil*.



**:eval-inside-yourself** form

■ The argument is a form that is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable is modified. This is intended to be used mainly for debugging.

**:funcall-inside-yourself** function &rest args

■ function is applied to args in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable is modified. This is a way of allowing callers to provide actions to be performed in an environment set up by the instance.

**:break**

■ **break** is called in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables.

## 19.12 Method combination

When a flavor has or inherits more than one method for an operation, they must be called in a specific sequence. The flavor system creates a function called a *combined method* which calls all the user-specified methods in the proper order. Invocation of the operation actually calls the combined method, which is responsible for calling the others.

For example, if the flavor *foo* has components and methods as follows:

---

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))

(defmethod (foo :before :hack) ...)
(defmethod (foo :after :hack) ...)

(defmethod (foo-mixin :before :hack) ...)
(defmethod (foo-mixin :after :hack) ...)

(defmethod (bar-mixin :before :hack) ...)
(defmethod (bar-mixin :hack) ...)

(defmethod (foo-base :hack) ...)
(defmethod (foo-base :after :hack) ...)
```

---

then the combined method generated looks like this (ignoring many details not related to this issue):



---

```
(defmethod (foo :combined :hack) (&rest args)
  (apply #'(:method foo :before :hack) args)
  (apply #'(:method foo-mixin :before :hack) args)
  (apply #'(:method bar-mixin :before :hack) args)
  (multiple-value-prog1
    (apply #'(:method bar-mixin :hack) args)
    (apply #'(:method foo-base :after :hack) args)
    (apply #'(:method foo-mixin :after :hack) args)
    (apply #'(:method foo :after :hack) args)))
```

---

This example shows the default style of method combination, the one described in the introductory parts of this chapter, called **:daemon** combination. Each style of method combination defines which *method types* it allows, and what they mean. **:daemon** combination accepts method types **:before** and **:after**, in addition to *untyped* methods; then it creates a combined method which calls all the **:before** methods, only one of the untyped methods, and then all the **:after** methods, returning the value of the untyped method. The combined method is constructed by a function much like a macro's expander function, and the precise technique used to create the combined method is what gives **:before** and **:after** their meaning.

Note that the **:before** methods are called in the order foo, foo-mixin, bar-mixin and foo-base. (foo-base does not have a **:before** method, but if it had one that one would be last.) This is the standard ordering of the components of the flavor foo; since it puts the base flavor last, it is called **:base-flavor-last** ordering. The **:after** methods are called in the opposite order, in which the base flavor comes first. This is called **:base-flavor-first** ordering.

Only one of the untyped methods is used; it is the one that comes first in **:base-flavor-last** ordering. An untyped method used in this way is called a *primary* method.

Other styles of method combination define their own method types and have their own ways of combining them. Use of another style of method combination is requested with the **:method-combination** option to **defflavor**. Here is an example which uses **:list** method combination, a style of combination that allows **:list** methods and untyped methods:

---

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))
(defflavor foo-base () ()
  (:method-combination (:list :base-flavor-last :win)))

(defmethod (foo :list :win) ...)
(defmethod (foo :win) ...)

(defmethod (foo-mixin :list :win) ...)
```



```
(defmethod (bar-mixin :list :win) ...)
(defmethod (bar-mixin :win) ...)

(defmethod (foo-base :win) ...)

; ; yielding this combined method

(defmethod (foo :combined :win) (&rest args)
  (list (apply #'(:method foo :list :win) args)
        (apply #'(:method foo-mixin :list :win) args)
        (apply #'(:method bar-mixin :list :win) args)
        (apply #'(:method foo :win) args)
        (apply #'(:method bar-mixin :win) args)
        (apply #'(:method foo-base :win) args)))
```

---

The **:method-combination** option in the **defflavor** for **foo-base** causes **:list** method combination to be used for the **:win** operation on all flavors that have **foo-base** as a component, including **foo**. The result is a combined method which calls all the methods, including all the untyped methods rather than just one, and makes a list of the values they return. All the **:list** methods are called first, followed by all the untyped methods; and within each type, the **:base-flavor-last** ordering is used as specified. If the **:method-combination** option said **:base-flavor-first**, the relative order of the **:list** methods would be reversed, and so would the untyped methods, but the **:list** methods would still be called before the untyped ones. **:base-flavor-last** is more often right, since it means that **foo**'s own methods are called first and **si:vanilla-flavors** methods (if it has any) are called last.

One method type, **:default**, has a standard meaning independent of the style of method combination, and can be used with any style.

Here are the standardly defined method combination styles:

#### **:daemon**

■ The default style of method combination. All the **:before** methods are called, then the primary (untyped) method for the outermost flavor that has one is called, then all the **:after** methods are called. The value returned is the value of the primary method.

#### **:daemon-with-or**

■ Like the **:daemon** method combination style, except that the primary method is wrapped in an **:or** special form with all **:or** methods. Multiple values can be returned from the primary method, but not from the **:or** methods (as in the **or** special form). This produces combined methods like the following:



---

```
(progn
  (foo-before-method)
  (multiple-value-prog1
    (or (foo-or-method)
        (foo-primary-method))
    (foo-after-method)))
```

---

This is useful primarily for flavors in which a mixin introduces an alternative to the primary method. Each **:or** method gets a chance to run before the primary method and to decide whether the primary method should be run or not; if any **:or** method returns a non-*nil* value, the primary method is not run (nor are the rest of the **:or** methods). Note that the ordering of the combination of the **:or** methods is controlled by the order keyword in the **:method-combination** option.

#### **:daemon-with-and**

■ Like **:daemon-with-or** except that it combines **:and** methods in an **and** special form. The primary method is run only if all of the **:and** methods return non-*nil* values.

#### **:daemon-with-override**

■ Like the **:daemon** method combination style, except an **or** special form is wrapped around the entire combined method with all **:override** typed methods before the combined method. This differs from **:daemon-with-or** in that the **:before** and **:after** daemons are run only if *none* of the **:override** methods returns non-*nil*. The combined method looks something like this:

---

```
(or (foo-override-method)
    (progn
      (foo-before-method)
      (multiple-value-prog1
        (foo-primary-method)
        (foo-after-method))))
```

---

#### **:progn**

■ Calls all the methods inside a **progn** special form. Only untyped and **:progn** methods are allowed. The combined method calls all the **:progn** methods and then all the untyped methods. The result of the combined method is whatever the last of the methods returns.



**:or**

■ Calls all the methods inside an **or** special form. This means that each of the methods is called in turn. Only untyped methods and **:or** methods are allowed; the **:or** methods are called first. If a method returns a non-*nil* value, that value is returned and none of the rest of the methods are called; otherwise, the next method is called. In other words, each method is given a chance to handle the message; if it doesn't want to handle the message, it can return *nil*, and the next method gets a chance to try.

**:and**

■ Calls all the methods inside an **and** special form. Only untyped methods and **:and** methods are allowed. The basic idea is much like **:or**; see above.

**:append**

■ Calls all the methods and appends the values together. Only untyped methods and **:append** methods are allowed; the **:append** methods are called first.

**:nconc**

■ Calls all the methods and **nconcs** the values together. Only untyped methods and **:nconc** methods are allowed, etc.

**:list**

■ Calls all the methods and returns a list of their returned values. Only untyped methods and **:list** methods are allowed, etc.

**:inverse-list**

■ Calls each method with one argument; these arguments are successive elements of the list that is the sole argument to the operation. Returns no particular value. Only untyped methods and **:inverse-list** methods are allowed, etc.

□ If the result of a **:list**-combined operation is sent back with an **:inverse-list**-combined operation, with the same ordering and with corresponding method definitions, each component flavor receives the value that came from that flavor.

**:pass-on**

■ NOTE: **:pass-on** method combination is not yet implemented in FRANZ LISP.

■ Calls each method on the values returned by the preceding one. The values returned by the combined method are those of the outermost call. The format of the declaration in the **defflavor** is:



---

```
(:method-combination (:pass-on (ordering . arglist)
                                'operation-names))
```

---

where ordering is **:base-flavor-first** or **:base-flavor-last**. arglist may include the **&aux** and **&optional** keywords.

☐ Only untyped methods and **:pass-on** methods are allowed. The **:pass-on** methods are called first.

### **:case**

■ With **:case** method combination, the combined method automatically does a **caseq** dispatch on the first argument of the operation, known as the *suboperation*. Methods of type **:case** can be used, and each one specifies one suboperation that it applies to. If no **:case** method matches the suboperation, the primary method, if any, is called.

---

```
(defflavor foo (a b) ()
  (:method-combination (:case :base-flavor-last :win)))

(defmethod (foo :case :win :a) () ;This method handles nd a-foo :win
:a):
  a)

(defmethod (foo :case :win :a*b) () ;This method handles nd a-foo
:win :a*b):
  (* a b))

(defmethod (foo :win) (suboperation) ;This method handles nd a-foo
:win :something-else):
  (list 'something-random suboperation))
```

---

**:case** methods are unusual in that one flavor can have many **:case** methods for the same operation, as long as they are for different suboperations.

☐ The suboperations **:which-operations**, **:operation-handled-p**, **:send-if-handles** and **:get-handler-for** are all handled automatically based on the collection of **:case** methods that are present.

■ NOTE: **:send-if-handles** and **:get-handler-for** are presently unimplemented in FRANZ LISP.

☐ Methods of type **:or** are also allowed. They are called just before the primary method, and if one of them returns a non-*nil* value, that is the value of the operation, and no more methods are called.



- ☐ Here is a list of all the method types recognized by the standard styles of method combination.

***no method type***

- If no type is given to **defmethod**, a primary method is created. This is the most common type of method.

***:before***

***:after***

- These are used for the before-daemon and after-daemon methods used by **:daemon** method combination.

***:default***

- If there are no untyped methods among any of the flavors being combined, then the **:default** methods (if any) are treated as if they were untyped. If there are any untyped methods, the **:default** methods are ignored.
- ☐ Typically a base-flavor defines some default methods for certain of the operations understood by its family. When using the default kind of method combination these default methods are suppressed if another component provides a primary method.

***:or***

***:and***

- These are used for **:daemon-with-or** and **:daemon-with-and** method combination. The **:or** methods are wrapped in an **or**, or the **:and** methods are wrapped in an **and**, together with the primary method, between the **:before** and **:after** methods.

***:override***

- Allows the features of **:or** method combination to be used together with daemons. If you specify **:daemon-with-override** method combination, you may use **:override** methods. The **:override** methods are executed first, until one of them returns non-*nil*. If this happens, that method's value(s) are returned and no more methods are used. If all the **:override** methods return *nil*, the **:before**, primary and **:after** methods are executed as usual.
- ☐ In typical usages of this feature, the **:override** method usually returns *nil* and does nothing, but in exceptional circumstances it takes over the handling of the operation.



**:or**  
**:and**  
**:progn**  
**:list**  
**:inverse-list**  
**:pass-on**  
**:append**  
**:nconc**

■ Each of these methods types is allowed in the method combination style of the same name. In those method combination styles, these typed methods work just like untyped ones, but all the typed methods are called before all the untyped ones. These method types can be used with any method combination style; they have standard meanings independent of the method combination style being used.

**:wrapper**

■ This is used internally by **defwrapper**.

**:combined**

■ This is used internally for automatically-generated *combined* methods.

□ The most common form of combination is **:daemon**. One thing may not be clear: when do you use a **:before** daemon and when do you use an **:after** daemon? In some cases the primary method performs a clearly-defined action and the choice is obvious: **:before :launch-rocket** puts in the fuel, and **:after :launch-rocket** turns on the radar tracking.

□ In other cases the choice can be less obvious. Consider the **:init** message, which is sent to a newly-created object. To decide what kind of daemon to use, we observe the order in which daemon methods are called. First the **:before** daemon of the instantiated flavor is called, then **:before** daemons of successively more basic flavors are called, and finally the **:before** daemon (if any) of the base flavor is called. Then the primary method is called. After that, the **:after** daemon for the base flavor is called, followed by the **:after** daemons at successively less basic flavors.

□ Now, if there is no interaction among all these methods, if their actions are completely independent, then it doesn't matter whether you use a **:before** daemon or an **:after** daemon. There is a difference if there is some interaction. The interaction we are talking about is usually done through instance variables; in general, instance variables are how the methods of different component flavors communicate with each other. In the case of the **:init** operation, the **init-plist** can be used as well. The important thing to remember is that no method knows beforehand which other flavors have been mixed in to form this flavor; a method cannot make any assumptions about how this flavor has been combined, and in what order the various components are mixed.



□ This means that when a **:before** daemon has run, it must assume that none of the methods for this operation have run yet. But the **:after** daemon knows that the **:before** daemon for each of the other flavors has run. So if one flavor wants to convey information to the other, the first one should *transmit* the information in a **:before** daemon, and the second one should *receive* it in an **:after** daemon. So while the **:before** daemons are run, information is *transmitted*; that is, instance variables get set up. Then, when the **:after** daemons are run, they can look at the instance variables and act on their values.

□ In the case of the **:init** method, the **:before** daemons typically set up instance variables of the object based on the init-plist, while the **:after** daemons actually do things, relying on the fact that all of the instance variables have been initialized by the time they are called.

□ The problems become most difficult when you are creating a network of instances of various flavors that are supposed to point to each other. For example, suppose you have flavors for *buffers* and *streams*, and each buffer should be accompanied by a stream. If you create the stream in the **:before :init** method for buffers, you can inform the stream of its corresponding buffer with an *init* keyword, but the stream may try sending messages back to the buffer, which is not yet ready to be used. If you create the stream in the **:after :init** method for buffers, there will be no problem with stream creation, but some other **:after :init** methods of other mixins may have run and made the assumption that there is to be no stream. The only way to guarantee success is to create the stream in a **:before** method and inform it of its associated buffer by sending it a message from the buffer's **:after :init** method. This scheme — creating associated objects in **:before** methods but linking them up in **:after** methods — often avoids problems, because all the various associated objects used by various mixins at least exist when it is time to make other objects point to them.

Since flavors are not hierarchically organized, the notion of levels of abstraction is not rigidly applicable. However, it remains a useful way of thinking about systems.

### 19.13 Implementation of flavors

An object that is an instance of a flavor is implemented as a vector with a **vprop** that is a *flavor descriptor*. Each instance variable is stored in a slot in the vector. The *flavor-descriptor* is also stored on the *si:flavor* property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling operations, and information for accessing the instance variables. The function (*describe-flavor* *flavor-name*) will print much of this information in readable format.

**defflavor** creates such a data structure for each flavor, and links them together according to the dependency relationships between flavors. Much of the information stored there, of course, is not computed until flavor-combination time.

A message is sent to an instance simply by calling it as a function, with the first argument being the operation. The evaluator looks up the operation in the dispatch hashtable



stored in the flavor descriptor for that flavor and obtains a handler function and a mapping table. It then binds `self` to the object, `sys:self-mapping-table` to the mapping table, and any instance variables that are declared to be special to the value cells in the instance. Finally, the handler function is called. If there is only one method to be invoked, the handler function is that method; otherwise it is an automatically-generated function, called the combined method, which calls the component methods appropriately. If there are wrappers, they are incorporated into the combined method.

The code body of each method function knows only about the instance variables declared for its flavor, and this set of instance variables is known when the defining `defmethod` is evaluated. However, the location of these instance variables within an instance of an arbitrary flavor containing that flavor is not known until flavor-combination time. The mapping table is used by a method to map the set of instance variables it knows about into slot offsets within `self`. If all the component methods invoked by the combined method derive from a single flavor, the mapping table obtained from the method dispatch hashtable is a vectori-word of byte offsets. If methods from more than one component flavor are invoked from the combined method, then the mapping table is an *alist* mapping component flavors to vectori-word mapping tables, and the combined method takes care of binding `si:self-mapping-table` appropriately.

Instance variables not explicitly declared special have lexical score in compiled method code. However, since the FRANZ LISP interpreter does not implement lexical closures, all instance variables in interpreted method code are implicitly special, whether or not they are so declared.

### 19.13.1 Order of definition

There is a certain amount of freedom to the order in which you do `defflavors`, `defmethods`, and `defwrappers`. This freedom is designed to make it easy to load programs containing complex flavor structures without having to do things in a certain order. It is considered important that not all the methods for a flavor need be defined in the same file. Thus the partitioning of a program into files can be along modular lines.

The rules for the order of definition are as follows.

Before a method can be defined (with `defmethod` or `defwrapper`) its flavor must have been defined (with `defflavor`). This makes sense because the system has to have a place to remember the method, and because it has to know the instance-variables of the flavor if the method is to be compiled.

When a flavor is defined (with `defflavor`) it is not necessary that all of its component flavors be defined already. This is to allow `defflavors` to be spread between files according to the modularity of a program, and to provide for mutually-dependent flavors. Methods can be defined for a flavor some of whose component flavors are not yet defined; however, in certain cases compiling those methods may produce a warning that an instance variable was declared special (because the system did not realize it was an instance variable). If this happens, you should fix the problem and recompile.

The methods automatically generated by the `:gettable-instance-variables` and `:settable-instance-variables` `defflavor` options are generated at the time the `defflavor`



is done.

The first time a flavor is instantiated, or when **compile-flavor-methods** is done, the system looks through all of the component flavors and gathers various information. At this point an error is signaled if not all of the components have been **defflavored**. This is also the time at which certain other errors are detected, for instance lack of a required instance-variable (see the **:required-instance-variables** option to **defflavor**). The set of instance variables is determined and their slots assigned within an instance. The combined methods are generated at this time also, unless they already exist and are correct. The flavor system tries very hard never to **redefun** a combined method unless its contents actually must change. This way, combined methods generated at compile time by **compile-flavor-methods** and then compiled will not get replaced by their equivalent interpreted code when flavor combination happens again at eval or load times.

After a flavor has been instantiated, it is possible to make changes to it. Such changes affect all existing instances if possible. This is described more fully immediately below.

### 19.13.2 Changing a flavor

You can change anything about a flavor at any time. You can change the flavor's general attributes by doing another **defflavor** with the same name. You can add or modify methods by doing **defmethods**. If you do a **defmethod** with the same flavor-name, operation (and suboperation if any), and (optional) method-type as an existing method, that method is replaced by the new definition.

These changes always propagate to all flavors that depend upon the changed flavor. Normally the system propagates the changes to all existing instances of the changed flavor and its dependent flavors. However, this is not possible when the flavor has been changed so drastically that the old instances would not work properly with the new flavor. This happens if you change the number of instance variables, which changes the size of an instance. It also happens if you change the order of the instance variables (and hence the storage layout of an instance), or if you change the component flavors (which can change several subtle aspects of an instance). The system does not keep a list of all the instances of each flavor, so it cannot find the instances and modify them to conform to the new flavor definition. Instead it gives you a warning message to the effect that the flavor was changed incompatibly and the old instances will not get the new version. The system leaves the old flavor data-structure intact (the old instances continue to point at it) and makes a new one to contain the new version of the flavor. If a less drastic change is made, the system modifies the original flavor data-structure, thus affecting the old instances that point at it. However, if you redefine methods in such a way that they only work for the new version of the flavor, then trying to use those methods with the old instances won't work.



## 19.14 Property list operations

It is often useful to associate a property list with an abstract object, for the same reasons that it is useful to have a property list associated with a symbol. This section describes a mixin flavor, `si:property-list-mixin`, that can be used as a component of any new flavor in order to provide that new flavor with a property list. For more details and examples, see the general discussion of property lists. The usual property list functionalities (`get`, `putprop`, etc.) are obtained by sending the instance the corresponding message. The contents of the property list can be initialized by providing a `:property-list` init option on the `init-plist` given to `Instantiate-flavor`.

`(si:property-list-mixin :get property-name)`

- Looks up the object's `property-name` property.

`(si:property-list-mixin :getl property-name-list)`

- Like the `:get` operation, except that the argument is a list of property names. The `:getl` operation searches down the property list until it finds a property whose property name is one of the elements of `property-name-list`. It returns the portion of the property list beginning with the first such property that it found. If it doesn't find any, it returns `nil`.

`(si:property-list-mixin :putprop value property-name)`

- Gives the object a `property-name` property of `value`.

`(si:property-list-mixin :remprop property-name)`

- Removes the object's `property-name` property, by splicing it out of the property list. It returns one of the cells spliced out, whose `car` is the former value of the property that was just removed. If there was no such property to begin with, the value is `nil`.

`(si:property-list-mixin :push-property value property-name)`

- The `property-name` property of the object should be a list (note that `nil` is a list and an absent property is `nil`). This operation sets the `property-name` property of the object to a list whose `car` is `value` and whose `cdr` is the former `property-name` property of the list. This is analogous to doing

---

```
(push value (get object property-name))
```

---



(si:property-list-mixin :property-list)

■ RETURNS the list of alternating property names and values that implements the property list.

(si:property-list-mixin :set-property-list list)

■ Sets the list of alternating property names and values that implements the property list to list.

## 19.15 Copying instances

There are no built-in techniques to copy instances because there are too many questions raised about what should be copied. These include:

- Do you or do you not send an *:init* message to the new instance? If you do, what init-plist options do you supply?
- If the instance has a property list, you should copy the property list (e.g. with **copy-list**) so that **putprop** or **remprop** on one of the instances does not affect the properties of the other instance.
- If the instance is a port connected to a network, some of the instance variables represent an agent in another host elsewhere in the network. Should the copy talk to the same agent, or should a new agent be constructed for it?
- If the instance is a port connected to a file, should copying the stream make a copy of the file or should it make another stream open to the same file? Should the choice depend on whether the file is open for input or for output?

In general, you can see that in order to copy an instance one must understand a lot about the instance. One must know what the instance variables mean so that the values of the instance variables can be copied if necessary. One must understand what relations to the external environment the instance has so that new relations can be established for the new instance. One must even understand what the general concept 'copy' means in the context of this particular instance, and whether it means anything at all.

Copying is a generic operation, whose implementation for a particular instance depends on detailed knowledge relating to that instance. Modularity dictates that this knowledge be contained in the instance's flavor, not in a *general copying function*. Thus the way to copy an instance is to send it a message, as in (send object :copy). It is up to you to implement the operation in a suitable fashion, such as

---

```
(defflavor foo (a b c) ()  
  (:inittable-instance-variables a b))  
  
(defmethod (foo :copy) ()  
  (make-instance 'foo :a a :b b))
```

---



The flavor system chooses not to provide any default method for copying an instance, and does not even suggest a standard name for the copying message, because copying involves so many semantic issues.

If a flavor supports the ***:reconstruction-init-plist*** operation, a suitable copy can be made by invoking this operation and passing the result to **make-instance** along with the flavor name. This is because the definition of what the ***:reconstruction-init-plist*** operation should do requires it to address all the problems listed above. Implementing this operation is up to you, and so is making sure that the flavor implements sufficient init keywords to transmit any information that is to be copied.







# A Summary of FRANZ LISP symbols







## A Summary of FRANZ LISP symbols

"~;" ... 5-18  
"~:@[" ... 5-18  
"~:{" ... 5-18  
"~}" ... 5-18  
"~;" ... 5-19  
"~:@[" ... 5-19  
"~:@{" ... 5-20  
"~:}" ... 5-20  
"~>" ... 5-21  
# ... 8-9  
#( ... 8-9  
#- ... 8-10  
#\ ... 8-10  
#+ ... 8-10  
#, ... 8-10  
#. ... 8-10  
#: ... 8-10  
#| ... 8-10  
|# ... 8-10  
? ... 15-6  
/ ... 16-7  
\_ ... 16-7  
\_<\_ ... 16-7  
? ... 16-8  
!! ... 16-9  
^ ... 16-9  
\$ ... 16-12  
(l) ... 16-14  
?? ... 16-18  
# c ... 8-11  
#/ c ... 8-11  
(l e1 ... em) ... 16-14  
(< 'fx\_arg1 'fx\_arg2) ... 3-4  
(= 'fx\_arg1 'fx\_arg2) ... 3-4  
(> 'fx\_arg1 'fx\_arg2) ... 3-4



(\# g\_com1 ...) ... 16-6  
 ?? [location-specifier] ... 13-2  
 (\_ pattern) ... 16-7  
 #+ when what ... 8-11  
 #- when what ... 8-12  
 (+ ['x\_arg1 ...]) ... 3-1  
 (/ ['x\_arg1 ...]) ... 3-2  
 (- ['x\_arg1 ...]) ... 3-2  
 (\* ['x\_arg1 ...]) ... 3-2  
 (<& 'x\_arg1 'x\_arg2) ... 3-4  
 (= & 'x\_arg1 'x\_arg2) ... 3-4  
 (>& 'x\_arg1 'x\_arg2) ... 3-4  
 (<=& 'x\_arg1 'x\_arg2) ... 3-4  
 (>=& 'x\_arg1 'x\_arg2) ... 3-4  
 0 ... 16-8  
 10 ... 16-9  
 (\$1 thru) ... 16-18  
 (\$1 thru \$2) ... 16-18  
 (\$1 to) ... 16-17  
 (\$1 to \$2) ... 16-17  
 (1+ 'x\_arg) ... 3-1  
 (1- 'x\_arg) ... 3-2  
 (a e1 ... em) ... 16-14  
 (abs 'n\_arg) ... 3-8  
 :abstract-flavor ... 19-30  
 (absval 'n\_arg) ... 3-8  
 :accessor-prefix ... 19-29  
 (acos 'fx\_arg) ... 3-5  
 (add ['n\_arg1 ...]) ... 3-1  
 (add1 'n\_arg) ... 3-1  
 ?add-function file1 fcn1 [fcn2 ...] ... 13-13  
 (addhash 'g\_key 'H\_htab 'g\_val) ... 2-25  
 ?add-prop file1 (sym1 ind1) [(sym2 ind2) ...] ... 13-13  
 (add-syntax-class 's\_synclass 'l\_properties) ... 7-13  
 ?add-var file1 var1 [var2 ...] ... 13-13  
 (aexplode 's\_arg) ... 2-21  
 (aexplodec 's\_arg) ... 2-21  
 (aexploden 's\_arg) ... 2-21  
 :after ... 19-40  
 :alias-flavor ... 19-29  
 \*all-flavor-names\* ... 19-16  
 (allocate 's\_type 'x\_pages) ... 6-1  
 (allsym 'ls\_arg) ... 2-16  
 (alphalessp 'st\_arg1 'st\_arg2) ... 2-16



**:alterant** [symbol | nil] ... 14-10  
**(alter-name** 's\_inst ['s\_slot1 'g\_val1...]) ... 14-15  
**:and** ... 19-38, 19-40, 19-41  
**(and** [g\_arg1 ...]) ... 4-1  
**:append** ... 19-38, 19-41  
**(append** 'l\_arg1 'l\_arg2 [ ... ]) ... 2-1  
**(append1** 'l\_arg1 'g\_arg2) ... 2-2  
**(apply** 'u\_func ['g\_arg1 ...] 'l\_args) ... 4-1  
**(apropos** 'st\_arg ['g\_package]) ... 4-2  
**(apropos-list** 'st\_arg ['g\_package]) ... 4-2  
**(arg** ['x\_numb]) ... 4-3  
**(argv** 'x\_argnumb) ... 6-1  
**:array** ... 14-13  
**(array** s\_name s\_type x\_dim1 ... x\_dimn) ... 2-27  
**(\*array** 's\_name 's\_type 'x\_dim1 ... 'x\_dimn) ... 2-27  
**(arraycall** s\_type 'as\_array 'x\_ind1 ... ) ... 2-28  
**(arraydims** 'a\_name) ... 2-28  
**(arrayp** 'g\_arg) ... 2-8, 2-27  
**(arrayref** 'a\_name 'x\_ind) ... 2-28  
**(ascii** 'x\_charnum) ... 2-13  
**(ash** 'x\_val 'x\_amt) ... 3-6  
**(asin** 'fx\_arg) ... 3-5  
**(assoc** 'g\_arg1 'l\_arg2) ... 2-31  
**(assq** 'g\_arg1 'l\_arg2) ... 2-31  
**(atan** 'fx\_arg1 'fx\_arg2) ... 3-5  
**(atom** 'g\_arg) ... 2-8  
**(attach** 'g\_x 'l\_l) ... 2-5  
**b** ... 15-2  
**(b e1 ... em)** ... 16-14  
**(baktrace)** ... 6-1  
**(bcdad** 's\_funcname) ... 2-35  
**(bcdp** 'g\_arg) ... 2-8  
**:before** ... 19-40  
**"~begin1,end1,begin2,end2,...;"** ... 5-19  
**(below** com x) ... 16-10  
**bf** pattern ... 16-12  
**(bf** pattern t) ... 16-12  
**(bi** n m) ... 16-16  
**(bignum-leftshift** bx\_arg x\_amount) ... 3-5  
**(bignum-to-list** 'b\_arg) ... 2-3  
**(bigp** 'g\_arg) ... 2-8  
**(bind** . coms) ... 16-21  
**bk** ... 16-9  
**BKA** ... 15-7



**BKC** ... 15-7  
**BKE** ... 15-7  
**BKF** ... 15-7  
**BKsuffix**; [n] [f] ... 15-7  
**BKV** ... 15-7  
**(bo n)** ... 16-16  
**(boole 'x\_key 'x\_v1 'x\_v2 ...)** ... 3-6  
**BOT** ... 15-5  
**(boundp 's\_name)** ... 2-16  
**:break** ... 19-34  
**(break [g\_message [g\_pred]])** ... 4-3  
**(\*break 'g\_pred 'g\_message)** ... 4-3  
**:but-first** ... 14-10  
**"~:@c"** ... 5-15  
**c** ... 15-1  
**?c [n]** ... 13-5  
**:callable-accessors** [arg] ... 14-9  
**(car 'l\_arg)** ... 2-3  
**:case** ... 19-39  
**(case 'g\_key-form l\_clause1 ...)** ... 4-3  
**(caseq 'g\_key-form l\_clause1 ...)** ... 4-3  
**(catch g\_exp [ls\_tag])** ... 4-4  
**(\*catch 'ls\_tag g\_exp)** ... 4-4  
**ccharacter** (*normal character*) ... 7-2  
**(c-declare l\_struct1 [l\_struct2 ...])** ... 18-12  
**(cdr 'l\_arg)** ... 2-3  
**(cerror 's\_continue-format-string 's\_error-format-string ['arg ...])** ... 4-11  
**cescape** (*single-character escape*) ... 7-5  
**(cfasl 'st\_file 'st\_entry 'st\_funcname ['st\_disc ['st\_library]])** ... 5-1  
**(cfasl 'st\_fname 'st\_cname 's\_lispname ['st\_discipline ['st\_libraries]])** ... 18-3  
**"c-function"** ... 18-5  
**(change \$ to e1 ... em)** ... 16-15  
**?changed** ... 13-13  
**(change-library-directory [keyword value ... ..])** ... E-2  
**(charcnt 'p\_port)** ... 5-2  
**(char-index 't\_string 'stx\_char)** ... 2-19  
**(char-rindex 't\_string 'stx\_char)** ... 2-19  
**(chdir 's\_path)** ... 6-2  
**cillegal** (*illegal character*) ... 7-6  
**cinfix-macro** (*infix character*) ... 7-6  
**cleft-bracket** (*left bracket*) ... 7-3  
**cleft-paren** (*left parenthesis*) ... 7-3  
**(cli:error 's\_format-string ['args])** ... 4-11  
**(close 'p\_port)** ... 5-2



(closure 'l\_vars 'g\_funcobj) ... 8-13  
 (closurep 'c\_closure) ... 2-9  
 (clrhash 'H\_htab) ... 2-25  
 cmacro (*character macro*) ... 7-5  
 cnumber (*digit*) ... 7-3  
 "~column,offset:t" ... 5-16  
 :combined ... 19-41  
 (command-line-args) ... 6-2  
 (comment [g\_arg ...]) ... 4-4  
 (compile-flavor-methods flavor-name ...) ... 19-22  
 (coms x1 ... xn) ... 16-19  
 (comsq com1 ... comn) ... 16-19  
 (concat ['stn\_arg1 ... ]) ... 2-12  
 (concatl 'l\_arg) ... 2-12  
 :conc-name [prefix] ... 14-5  
 (cond [l\_clause1 ...]) ... 4-5  
 (cons 'g\_arg1 'g\_arg2) ... 2-1  
 :constructor [name [init-list]] ... 14-8  
 (copy \$1 to com . \$2) ... 16-16  
 (copy 'g\_arg) ... 2-35  
 (copyint\* 'x\_arg) ... 2-35  
 (copysymbol 's\_arg 'g\_pred) ... 2-13  
 (copy-symbol 's\_arg 'g\_pred) ... 2-13  
 (cos 'fx\_angle) ... 3-5  
 (cp com . \$) ... 16-16  
 cpackage (*package marker*) ... 7-3  
 cperiod (*dotted-pair separator*) ... 7-4  
 (cprintf 'st\_format 'xfst\_val ['p\_port]) ... 5-2  
 (cpy1 'xvt\_arg) ... 2-35  
 (c..r 'lh\_arg) ... 2-4  
 cright-bracket (*right bracket*) ... 7-4  
 cright-paren (*right parenthesis*) ... 7-3  
 cseparator (*token separator*) ... 7-4  
 csign (*leading sign*) ... 7-3  
 csingle-character-symbol (*single-character symbol*) ... 7-5  
 csingle-infix-macro (*single infix character*) ... 7-6  
 csingle-macro (*conditional character macro*) ... 7-5  
 csingle-quote (*quote*) ... 7-4  
 csingle-splicing-macro (*conditional splicing character macro*) ... 7-6  
 csplicing-macro (*splicing character macro*) ... 7-5  
 cstring-delimiter (*string quote*) ... 7-5  
 csymbol-delimiter (*multi-character escape*) ... 7-4  
 (cvttofranzlisp) ... 4-5  
 (cvttointlisp) ... 4-5



(cvttomacclisp) ... 4-5  
 (cvtoucillisp) ... 4-5  
 (cxr 'x\_ind 'h\_hunk) ... 2-30  
 d ... 15-2  
 :daemon ... 19-36  
 :daemon-with-and ... 19-37  
 :daemon-with-or ... 19-36  
 :daemon-with-override ... 19-37  
 ?debug ... 13-2  
 ?debug full ... 13-2  
 (debug s\_msg) ... 4-6  
 (debug [s\_msg]) ... 15-5  
 (debugging 'g\_arg) ... 4-6  
 (declare [g\_arg ...]) ... 4-6  
 (def s\_name (s\_type l\_argl g\_exp1 ...)) ... 4-6  
 :default ... 19-40  
 :default-handler ... 19-28  
 :default-init-plist ... 19-25  
 :default-pointer ... 14-8  
 (defcmacro s\_name l\_arg g\_exp1 ...) ... 4-6  
 (defflavor flavor-name ([var] ...) ([flav] ...) [option] ...) ... 19-16  
 #define **NAMESIZE** 3072 ... E-1  
 #define **NAMINC** 50 ... E-1  
 #define **TTSIZE** 6120 ... E-1  
 #define **XstackSize** ... E-1  
 (defmacro s\_name l\_arg g\_exp1 ...) ... 4-6  
 defmacro-for-compiling ... B-2  
 (defmethod (flavor-name [ method-type ] operation) lambda-list [form] ...) ... 19-17  
 (defprop ls\_name g\_val g\_ind) ... 2-33  
 (defsetf s\_name l\_setfvars 'g\_body) ... 14-2  
 (defsharp g\_key l\_arglist g\_exp1 ...) ... 8-9  
 (defstruct sl\_nameargs sl\_slotdescript1 [...sl\_slotdescripti]) ... 14-5  
 (defsubst s\_name l\_list g\_form [...]) ... 4-6  
 (defun s\_name [s\_mtype] ls\_argl g\_exp1 ... ) ... 4-7  
 (defvar s\_variable ['g\_init]) ... 4-8  
 (defwhopper (flavor-name operation) lambda-list &body body) ... 19-21  
 (defwrapper (flavor-name operation) lambda-list &body body) ... 19-20  
 delete ... 16-14  
 (delete . \$) ... 16-14  
 (delete 'g\_val 'l\_list ['x\_count]) ... 2-5  
 (delq 'g\_val 'l\_list ['x\_count]) ... 2-6  
 (deref 'x\_addr) ... 6-2  
 :describe ... 19-33  
 (describe-cs v\_item) ... 18-16



(describe-flavor 'flavor-name) ... 19-23  
 (desetq sl\_pattern1 'g\_exp1 [... ...]) ... 2-19  
 (diff ['n\_arg1 ... ]) ... 3-2  
 (difference ['n\_arg1 ... ]) ... 3-2  
 "~digitse" ... 5-14  
 "~digitsf" ... 5-14  
 displace-macros ... B-2, 13-7  
 (Divide 'i\_dividend 'i\_divisor) ... 3-2  
 ?dn [n] ... 13-4  
 DN n ... 15-6  
 DNFN n ... 15-6  
 (do l\_vrbs l\_test g\_exp1 ...) ... 4-8  
 (do\* l\_vrbs l\_test g\_exp1 ...) ... 4-9  
 (do s\_name g\_init g\_repeat g\_test g\_exp1 ...) ... 4-9  
 (do-all-symbols l\_illist l\_body) ... 17-9  
 :documentation ... 19-32  
 (do-external-symbols l\_illist l\_body) ... 17-9  
 (dolist (s\_var l\_form g\_resultform) g\_exp1 ...) ... 4-9  
 (do-symbols l\_illist l\_body) ... 17-9  
 (dotimes (s\_var i\_countform g\_resultform) g\_exp1 ...) ... 4-9  
 "double-c-function" ... 18-5  
 (double-to-float 'f\_flo) ... 18-16  
 (drain ['p\_port]) ... 5-3  
 (dremove 'g\_val 'l\_list ['x\_count]) ... 2-6  
 (dsbst 'g\_x 'g\_y 'l\_s) ... 2-7  
 (dtp 'g\_arg) ... 2-3, 2-9  
 (dumplisp s\_name) ... 6-2  
 :dumpname s\_dumpname ... E-2  
 e ... 15-2, 16-19  
 (e x) ... 16-19  
 (e x t) ... 16-19  
 editcomsl ... 16-22  
 (editf s\_x1 ...) ... 16-5  
 (editflndp x pat nil) ... 16-6  
 (editfns s\_x [ g\_coms1 ... ]) ... 16-6  
 (editp s\_x) ... 16-5  
 (editracefn s\_com) ... 16-6  
 (editv s\_var [ g\_com1 ... ]) ... 16-5  
 (embed \$ in . x) ... 16-15  
 (Emuldiv 'x\_fact1 'x\_fact2 'x\_addn 'x\_divisor) ... 3-3  
 environment ... B-2  
 (environment [l\_when1 l\_what1 l\_when2 l\_what2 ...]) ... 4-10  
 (environment-lmlisp [l\_when1 l\_what1 l\_when2 l\_what2 ...]) ... 4-10  
 (environment-maclisp [l\_when1 l\_what1 l\_when2 l\_what2 ...]) ... 4-10



EOF ... 13-4  
 (eq 'g\_arg1 'g\_arg2) ... 2-10  
 (eq1 'g\_arg1 'g\_arg2) ... 2-11  
 (eqstr 'g\_arg1 'g\_arg2) ... 2-11  
 (equal 'g\_arg1 'g\_arg2) ... 2-11  
 ER%all ... B-1  
 ER%brk ... B-1  
 ER%err ... B-1  
 ER%misc ... B-2  
 (err ['s\_value [nil]]) ... 4-11  
 errlist ... B-2  
 (error ['s\_message1 ['s\_message2]]) ... 4-11  
 errport ... B-3  
 (errset g\_expr [s\_flag]) ... 4-11  
 ER%tpl ... B-2  
 ER%undef ... B-2  
 (escape-aexploden 's\_arg) ... 2-21  
 (escape-exploden 'g\_arg) ... 2-20  
 ?ev symbol ... 13-4  
 (eval 'g\_val ['x\_bind-pointer]) ... 4-12  
 (evalframe 'x\_pdlpointer) ... 4-12  
 evalhook ... B-3  
 (evalhook 'g\_form 'su\_evalfunc ['su\_funcallfunc]) ... 4-13  
 :eval-inside-yourself form ... 19-34  
 (eval-when l\_time g\_exp1 ...) ... 6-2  
 :eval-when when-keywords ... 14-10  
 (evenp 'x\_arg) ... 3-3  
 (exec s\_arg1 ...) ... 6-2  
 (exece 's\_fname ['l\_args ['l\_envir]]) ... 6-3  
 (exit ['x\_code]) ... 6-3  
 (exp 'fx\_arg) ... 3-8  
 (explode 'g\_arg) ... 2-20  
 (explodec 'g\_arg) ... 2-20  
 (exploden 'g\_arg) ... 2-20  
 (export 'ls\_syms ['k\_package]) ... 17-7  
 (expt 'n\_base 'n\_power) ... 3-8  
 (extract \$1 from \$2) ... 16-15  
 (f= expression x) ... 16-11  
 f pattern ... 16-11  
 (f pattern) ... 16-11  
 (f pattern n) ... 16-11  
 (f pattern nil) ... 16-11  
 (f pattern t) ... 16-11  
 (fact 'x\_arg) ... 3-9



(fake 'x\_addr) ... 6-3  
 "~:[false~;true~<]" ... 5-18  
 (fasl 'st\_name ['st\_mapf ['g\_warn]]) ... 5-3  
 ?fast ... 13-3  
 (fboundp 's\_arg) ... 2-18  
 (fclosure 'l\_vars 'g\_funobj) ... 8-15  
 (fclosure-alist 'v\_fclosure) ... 8-15  
 (fclosure-function 'v\_fclosure) ... 8-15  
 (fclosure-list 'l\_vars1 'g\_fcnoobj1 [... ...]) ... 8-15  
 (fclosurep 'c\_closure) ... 2-10  
 (fclosurep 'v\_fclosure) ... 8-15  
 (feature-present 'g\_exp) ... 6-10  
 (ffasl 'st\_fname 'st\_cname 's\_lispname ['st\_discipline ['st\_libraries]]) ... 18-3  
 file-backup-append ... 13-14  
 file-backup-prepend ... 13-14  
 ?filein [name1 name2 ...] ... 13-12  
 (fileopen 's\_filename 's\_mode)) ... 5-3  
 ?fileout [name1 name2 ...] ... 13-13  
 (filepos 'p\_port ['x\_pos]) ... 5-3  
 (filestat 'st\_filename) ... 5-3  
 (filestat-atime 'v\_filestat) ... 5-4  
 (filestat-ctime 'v\_filestat) ... 5-4  
 (filestat-dev 'v\_filestat) ... 5-4  
 (filestat-gid 'v\_filestat) ... 5-4  
 (filestat-lno 'v\_filestat) ... 5-4  
 (filestat-mode 'v\_filestat) ... 5-4  
 (filestat-mtime 'v\_filestat) ... 5-4  
 (filestat-nlink 'v\_filestat) ... 5-4  
 (filestat-rdev 'v\_filestat) ... 5-4  
 (filestat-size 'v\_filestat) ... 5-4  
 (filestat-type 'v\_filestat) ... 5-4  
 (filestat-uid 'v\_filestat) ... 5-4  
 ?filestatus [filen1 filen2 ...] ... 13-13  
 (fillarray 's\_array 'l\_itms) ... 2-29  
 (find-all-symbols 'st\_name) ... 17-9  
 (flnd-package 's\_name) ... 17-5  
 (flnd-symbol t\_string ['k\_package]) ... 17-7  
 "~first,second,third:~" ... 5-22  
 (fix 'n\_arg) ... 3-9  
 :fixnum ... 14-14  
 (fixp 'g\_arg) ... 3-3  
 (flatc 'g\_form ['x\_max]) ... 5-4  
 (flatsize 'g\_form ['x\_max]) ... 5-4  
 (flavor-allows-init-keyword-p 'flavor-name 'keyword) ... 19-23



(float 'n\_arg) ... 3-9  
 float-format ... B-3  
 (floatp 'g\_arg) ... 3-3  
 (float-to-double 'x\_fix) ... 18-16  
 foo ... 11-1  
 (foo break) ... 11-1  
 (foo evalin expression) ... 11-2  
 (foo evalinout expression) ... 11-2  
 (foo evalout expression) ... 11-2  
 (foo evfcn evfunc) ... 11-3  
 (foo if expression) ... 11-2  
 (foo ifnot expression) ... 11-2  
 (foo lprint) ... 11-2  
 (foo printargs prfunc) ... 11-3  
 (foo printres prfunc) ... 11-3  
 (foo traceenter tefunc) ... 11-3  
 (foo traceexit txfunc) ... 11-3  
 (fork) ... 6-3  
 (format 'p\_port 's\_ctrl ['g\_arg ...]) ... 5-12  
 (freturn 'x\_pdl-pointer 'g\_retval) ... 4-13  
 (fs pattern1 ... patternn) ... 16-11  
 (fseek 'p\_port 'x\_offset 'x\_flag) ... 5-5  
 (funcall 's\_instance 's\_message ['argument ...]) ... 19-22  
 (funcall 'u\_func ['g\_arg1 ...]) ... 4-13  
 funcallhook ... B-3  
 (funcallhook 'l\_form 'su\_funcallfunc ['su\_evalfunc]) ... 4-14  
 :funcall-inside-yourself function &rest args ... 19-34  
 (funcall-self 's\_message ['argument] ...) ... 19-22  
 "function" ... 18-4  
 (function u\_func) ... 4-14  
 g ... 15-2  
 (gc) ... 6-4  
 (gcafter s\_type) ... 6-4  
 (gcbefore s\_type) ... 6-4  
 \$gccount\$ ... B-1  
 gcdisable ... B-3  
 \$gcprint ... B-1  
 (gensym ['s\_leader]) ... 2-14  
 (gentemp ['s\_name ['k\_package]]) ... 2-14  
 (get 'ls\_name 'g\_ind) ... 2-33  
 (getaccess 'a\_array) ... 2-28  
 (getaddress 's\_entry1 's\_binder1 'st\_discipline1 [... ..]) ... 2-35  
 (getaddress 'st\_cname 's\_lispname ['st\_discipline] ...) ... 18-3  
 (getaux 'a\_array) ... 2-28



(getchar 's\_arg 'x\_index) ... 2-18  
 (getcharn 's\_arg 'x\_index) ... 2-18  
 (getd 's\_arg) ... 2-18  
 (getdata 'a\_array) ... 2-28  
 (getdelta 'a\_array) ... 2-28  
 (getdisc 'y\_bcd) ... 2-31  
 (getentry 'y\_bcd) ... 2-31  
 (getenv 'st\_name) ... 6-4  
 (get-handler-for 'object 'operation) ... 19-23  
 :get-handler-for operation ... 19-33  
 (gethash 'g\_key 'H\_htab [ 'g\_defval ]) ... 2-25  
 (getl 'ls\_name 'l\_indicators) ... 2-33  
 (getlength 'a\_array) ... 2-28  
 (get\_pname 's\_arg) ... 2-18  
 (getsyntax 's\_symbol) ... 7-13  
 :gettable-instance-variables ... 19-24  
 (go g\_labexp) ... 4-14  
 (greaterp ['n\_arg1 ...]) ... 3-4  
 H ... 15-6  
 (haipart bx\_number x\_bits) ... 3-5  
 (hash-table-count 'H\_htab) ... 2-26  
 (hash-table-p 'H\_arg) ... 2-10  
 (haulong 'bx\_number) ... 3-5  
 HELP ... 15-6  
 (help sx\_arg) ... 4-14  
 ?help [topic] ... 13-2  
 ?his[tory] [r] ... 13-3  
 :hunk ... 14-13  
 (hunk 'g\_val1 ['g\_val2 ... 'g\_valn]) ... 2-30  
 (hunkp 'g\_arg) ... 2-9  
 (hunksize 'h\_arg) ... 2-31  
 (hunk-to-list 'h\_hunk) ... 2-30  
 (i c x1 ... xn) ... 16-19  
 ibase ... B-3  
 (if 'g\_a 'g\_b) ... 4-15  
 (if 'g\_a 'g\_b 'g\_c ...) ... 4-15  
 (if 'g\_a then 'g\_b [...] [elseif 'g\_c then 'g\_d ...] [else 'g\_e [...]]) ... 4-15  
 (if 'g\_a then 'g\_b [...] [elseif 'g\_c thenret] [else 'g\_d [...]]) ... 4-15  
 (if pred form1 [form2]) ... 4-5  
 (if x) ... 16-20  
 (if x coms1) ... 16-20  
 (if x coms1 coms2) ... 16-20  
 (implode 'l\_arg) ... 2-12  
 (implodes 'l\_arg) ... 2-12



(Import 'lg\_sym ['k\_package]) ... 17-8  
 (include s\_filename) ... 6-4  
 :include structure [component | (component initial-value) ...] ... 14-6  
 :included-flavors ... 19-27  
 (includef 's\_filename) ... 6-4  
 (includef-if 'g\_predicate s\_filename) ... 6-5  
 (include-lf 'g\_predicate s\_filename) ... 6-4  
 (inlle 's\_filename) ... 5-5  
 :init init-plist ... 19-19  
 :initable-instance-variables ... 19-24  
 :initial-offset offset ... 14-9  
 :init-keywords ... 19-25  
 (inltsym 'ls\_arg1 ...) ... 2-15  
 (in-package 'st\_pckname [:nicknames 'l\_nicklist] [:use 'p\_usepack]) ... 17-5  
 (insert e1 ... em after . \$) ... 16-14  
 (insert e1 ... em before . \$) ... 16-14  
 (insert e1 ... em for . \$) ... 16-15  
 (insert 'g\_object 'l\_list 'u\_comparefn 'g\_nodups) ... 2-6  
 (instancep 'object) ... 19-19  
 (instantiate-flavor 'flavor-name 'init-plist ['send-init-message-p 'return-unhandled-  
     keywords 'area]) ... 19-18  
 "integer-function" ... 18-5  
 (integer-length 'bx\_number) ... 3-5  
 (intern 's\_arg ['k\_package]) ... 2-13  
 (intern t\_string ['k\_package]) ... 17-6  
 :inverse-list ... 19-38, 19-41  
 (\*invmod 'x\_number 'x\_modulus) ... 3-3  
 (keywordp 's\_sym) ... 2-9  
 (kwote 'g\_arg) ... 2-36  
 ?l [n] ... 13-5  
 (last 'l\_arg) ... 2-4  
 (lc . \$) ... 16-12  
 (lcl . \$) ... 16-12  
 (lconc 'l\_ptr 'l\_x) ... 2-34  
 ?ld [file1 file2 ...] ... 13-3  
 (ldlff 'l\_x 'l\_y) ... 2-5  
 \$ldprint ... B-1  
 :left-margin x\_lmar ... 5-11  
 (length 'l\_arg) ... 2-3  
 (lessp ['n\_arg1 ...]) ... 3-4  
 (let\* l\_args g\_exp1 ... g\_expn) ... 4-16  
 (let l\_args g\_exp1 ... g\_exprn) ... 4-15  
 (let-closed argument-list function-body) ... 8-14  
 (lexpr-funcall 'g\_function ['g\_arg1 ...] 'l\_argn) ... 4-16



(lexpr-funcall-self 's\_message ['argument ...] 'list-of-arguments) ... 19-22  
 (lexpr-send 's\_object 's\_message ['argument ...] 'list-of-arguments) ... 19-8  
 (lexpr-send-self 's\_message ['argument ...] 'list-of-arguments) ... 19-22  
 (ll n) ... 16-16  
 :libdir t\_libdir ... E-2  
 line1 ... B-3  
 lisp-library-directory ... B-3  
 lisp-object-directory ... B-3  
 :list ... 14-13  
 :list\* ... 14-13  
 :list ... 19-38, 19-41  
 (llst ['g\_arg1 ... ]) ... 2-1  
 (llst\* ['g\_arg1 ... ]) ... 2-1  
 (llst-all-packages) ... 17-6  
 (llstarray 'sa\_array ['x\_elements]) ... 2-28  
 (llstify 'x\_count) ... 4-16  
 (llstp 'g\_arg) ... 2-3, 2-9  
 (llst-to-blgnum 'l\_ints) ... 2-3  
 (lltatom 'g\_arg) ... 2-9  
 (lo n) ... 16-17  
 (load 's\_filename ['st\_map ['g\_warn]]) ... 5-5  
 load-most-recent ... B-3  
 (log 'fx\_arg) ... 3-9  
 (logand ['n\_arg1 ...]) ... 3-7  
 (logandc1 'n\_arg1 'n\_arg2) ... 3-7  
 (logandc2 'n\_arg1 'n\_arg2) ... 3-7  
 (logbitp 'x\_index 'n\_number) ... 3-7  
 (logcount 'n\_number) ... 3-8  
 (logeqv ['n\_arg1 ...]) ... 3-7  
 (loglor ['n\_arg1 ...]) ... 3-7  
 (loglorc1 'n\_arg1 'n\_arg2) ... 3-8  
 (loglorc2 'n\_arg1 'n\_arg2) ... 3-8  
 (lognand 'n\_arg1 'n\_arg2) ... 3-8  
 (lognor 'n\_arg1 'n\_arg2) ... 3-8  
 (lognot 'n\_arg) ... 3-7  
 (logtest 'n\_arg1 'n\_arg2) ... 3-8  
 (logxor ['n\_arg1 ...]) ... 3-7  
 (lp . coms) ... 16-20  
 (lpq . coms) ... 16-20  
 (lsh 'x\_val 'x\_amt) ... 3-6  
 (lsubst 'l\_x 'g\_y 'l\_s) ... 2-7  
 ?m ... 13-6  
 (m bp bk up p) ... 16-21  
 (m c . coms) ... 16-21



(macroexpand 'g\_form) ... 2-35  
 (makefn form args n) ... 16-23  
 (makefn form args n m) ... 16-23  
 (make-hash-table [keyword value ... ..]) ... 2-24  
 (make-instance 'flavor-name ['init-option 'value] ... ) ... 19-18  
 (make-name ['s\_slot1 'g\_val1...]) ... 14-15  
 (make-package 'st\_pkname [:nicknames 'l\_names] [:use 'l\_packs]) ... 17-5  
 (makereadtable ['s\_flag]) ... 5-6  
 (\*makhunk 'x\_arg) ... 2-30  
 (makhunk 'xl\_arg) ... 2-30  
 (maknam 'l\_arg) ... 2-12  
 (maknum 'g\_arg) ... 6-5  
 (makunbound 's\_arg) ... 2-20  
 (map 'u\_func 'l\_arg1 ...) ... 4-16  
 (mapc 'u\_func 'l\_arg1 ...) ... 4-16  
 (mapcan 'u\_func 'l\_arg1 ...) ... 4-16  
 (mapcar 'u\_func 'l\_arg1 ...) ... 4-16  
 (mapcon 'u\_func 'l\_arg1 ...) ... 4-16  
 (maphash 'u\_fun 'H\_htab [ 'g\_rest ]) ... 2-25  
 (maplist 'u\_func 'l\_arg1 ...) ... 4-17  
 mark ... 16-7  
 (marray 'g\_data 's\_access 'g\_aux 'x\_length 'x\_delta) ... 2-27  
 (max 'n\_arg1 ... ) ... 3-9  
 (mbd x) ... 16-15  
 (member 'g\_arg1 'l\_arg2) ... 2-12  
 (memq 'g\_arg1 'l\_arg2) ... 2-12  
 (merge 'l\_data1 'l\_data2 'u\_comparefn) ... 2-6  
**:method-combination** ... 19-30  
 (mfunction t\_entry 's\_disc") ... 4-17  
 (min 'n\_arg1 ... ) ... 3-9  
 "~mincolw,colincr,minpad,padchar:@<" ... 5-21  
 "~mincolw,padchard" ... 5-14  
 "~mincolw,padcharo" ... 5-14  
 (minus 'n\_arg) ... 3-2  
 (minusp 'g\_arg) ... 3-4  
**:mixture** defflavor ... 19-30  
 (mod 'i\_dividend 'i\_divisor) ... 3-9  
 (\*mod 'x\_dividend 'x\_divisor) ... 3-9  
 \*modules\* ... 17-10  
 (move \$1 to com . \$2) ... 16-15  
 (msg [l\_option ...] ['g\_msg ...]) ... 5-6  
 (multiple-value-bind 'l\_varlist 'g\_values-form 'g\_form1 [ 'g\_form2 ... ]) ... 4-21  
 (multiple-value-call 'u\_fun 'g\_form1 [ 'g\_form2 ... ]) ... 4-20  
 (multiple-value-list 'g\_form) ... 4-21



**(multiple-value-prog1 'g\_form1 [ 'g\_form2 ... ]) ... 4-21**  
**(multiple-value-setq l\_varlist 'g\_form) ... 4-21**  
*multiple-values-limit ... B-4*  
**(mv com . \$) ... 16-16**  
**-n ... 16-8**  
**n ... 16-8**  
**(n) ... 16-13**  
**(n e1 ... em) ... 16-13**  
**(-n e1 ... em) ... 16-13**  
**(n e1 ... em) ... 16-14**  
**n number ... 15-2**  
**:named ... 14-6, 14-8**  
**:named-array ... 14-13**  
**:named-hunk ... 14-13**  
**:named-list ... 14-13**  
**:named-vector ... 14-14**  
**:nconc ... 19-38, 19-41**  
**(nconc 'l\_arg1 'l\_arg2 ['l\_arg3 ...]) ... 2-7**  
**(ncons 'g\_arg) ... 2-1**  
**(neq 'g\_x 'g\_y) ... 2-10**  
**(nequal 'g\_x 'g\_y) ... 2-10**  
**(newsym 's\_arg) ... 2-15**  
**(new-vector 'x\_size ['g\_fill ['g\_prop]]) ... 2-21**  
**(new-vectori-byte 'x\_size ['g\_fill ['g\_prop]]) ... 2-21**  
**(new-vectori-double 'x\_size ['g\_fill ['g\_prop]]) ... 2-21**  
**(new-vectori-float 'x\_size ['g\_fill ['g\_prop]]) ... 2-21**  
**(new-vectori-long 'x\_size ['g\_fill ['g\_prop]]) ... 2-21**  
**(new-vectori-word 'x\_size ['g\_fill ['g\_prop]]) ... 2-21**  
**nex ... 16-10**  
**(nex x) ... 16-10**  
**nil ... B-4**  
**nil ... 16-22**  
**no method type ... 19-40**  
**(not 'g\_arg) ... 2-11**  
**:no-vanilla-flavor ... 19-28**  
**(nreconc 'l\_arg 'g\_arg) ... 2-8**  
**(nreverse 'l\_arg) ... 2-8**  
**(nstring-capitalize 't\_string) ... 2-17**  
**(nstring-downcase 't\_string) ... 2-17**  
**(nstring-upcase 't\_string) ... 2-17**  
**(nth \$) ... 16-9**  
**(nth n) ... 16-9**  
**(nth 'x\_index 'l\_list) ... 2-4**  
**(nthcdr 'x\_index 'l\_list) ... 2-4**



(nthchar 's\_arg 'x\_index) ... 2-18  
(nthelem 'x\_arg1 'l\_arg2) ... 2-4  
(null 'g\_arg) ... 2-11  
"~number%" ... 5-15  
"~number&" ... 5-16  
"~number:|" ... 5-16  
"~number~" ... 5-16  
"~number:\*" ... 5-18  
"~numberg" ... 5-18  
(numberp 'g\_arg) ... 3-3  
"~numberx" ... 5-16  
(numbp 'g\_arg) ... 3-3  
(nwritn ['p\_port]) ... 5-6  
lnx ... 16-9  
nx ... 16-9  
(nx n) ... 16-9  
#O ... 8-10  
#o ... 8-10  
:objdir t\_objdir ... E-2  
(oblist) ... 4-17  
(oddp 'x\_arg) ... 3-3  
OK ... 15-6  
ok ... 16-22  
(oldsym 's\_arg) ... 2-15  
(onep 'g\_arg) ... 3-3  
:operation-handled-p operation ... 19-33  
(opval 's\_arg ['g\_newval]) ... 6-5  
:or ... 19-38, 19-40, 19-41  
(or [g\_arg1 ... ]) ... 4-17  
:ordered-instance-variables ... 19-28  
(orf pattern1 ... patternn) ... 16-12  
(orr coms1 ... comsn) ... 16-20  
(outfile 's\_filename ['st\_type]) ... 5-6  
:outside-accessible-instance-variables ... 19-29  
:override ... 19-40  
"~:@p" ... 5-16  
p ... 15-2  
P ... 15-5  
/p ... 16-7  
p ... 16-7  
(p foo) ... 16-8  
(p m) ... 16-7  
(p m n) ... 16-8  
?p [n1 n2 part] ... 13-5



**?P** [n1 n2 part] ... 13-5  
**\*package\*** ... B-4  
**\*package\*** ... 17-5  
**(package-name 'k\_pack)** ... 17-6  
**(package-nicknames 'k\_pkname)** ... 17-6  
**(packagep 'k\_package)** ... 2-9  
**(package-shadowing-symbols 'k\_pack)** ... 17-6  
**(package-used-by-list 'k\_pack)** ... 17-6  
**(package-use-list 'k\_pack)** ... 17-6  
**:pass-on** ... 19-38, 19-41  
**(patom 'g\_exp ['p\_port])** ... 5-7  
**piport** ... B-4  
**(plist 's\_name)** ... 2-33  
**(plus ['n\_arg1 ...])** ... 3-1  
**(plusp 'n\_arg)** ... 3-3  
**(pntlen 'xfs\_arg)** ... 5-7  
**?pop** ... 13-4  
**(pop 'l\_stack ['g\_into])** ... 4-17  
**poport** ... B-4  
**:port p\_port** ... 5-11  
**(portp 'g\_arg)** ... 5-7  
**?pp** ... 13-4  
**PP** ... 15-5  
**pp** ... 16-8  
**pp\*** ... 16-8  
**(pp [l\_option] s\_name1 ...)** ... 5-7  
**(pp-form 'g\_form ['p\_port])** ... 5-8  
**?ppp [part]** ... 13-5  
**?PPP [part]** ... 13-6  
**(primep 'x\_arg)** ... 3-4  
**(princ 'g\_arg ['p\_port])** ... 5-7  
**prinlength** ... B-4  
**prinlevel** ... B-4  
**:print** control-string [control-args ...] ... 14-12  
**(print 'g\_arg ['p\_port])** ... 5-7  
**:print-function** u\_func ... 14-12  
**:print-self** stream prindepth escape-p ... 19-33  
**(probe-file 'st\_file)** ... 5-8  
**(process s\_pgrm [s\_frompipe s\_topipe])** ... 6-6  
**(\*process 'st\_command ['g\_readp ['g\_writep]])** ... 6-5  
**(\*process-receive 'st\_command)** ... 6-6  
**(\*process-send 'st\_command)** ... 6-6  
**(product ['n\_arg1 ...])** ... 3-2  
**(prof-end)** ... 6-13



(prof-report) ... 6-13  
 (prof-start) ... 6-13  
 (prog l\_vrbls g\_exp1 ...) ... 4-18  
 (prog1 'g\_exp1 ['g\_exp2 ...]) ... 4-18  
 (prog2 'g\_exp1 'g\_exp2 ['g\_exp3 ...]) ... 4-18  
 :progn ... 19-37, 19-41  
 (progn 'g\_exp1 ['g\_exp2 ...]) ... 4-18  
 (progv 'l\_locv 'l\_initv g\_exp1 ...) ... 4-18  
 (provide 's\_name) ... 17-10  
 ?prt ... 13-3  
 (ptime) ... 6-6  
 ptport ... B-4  
 (ptr 'g\_arg) ... 2-36  
 (purcopy 'g\_exp) ... 4-18  
 (purep 'g\_exp) ... 4-18  
 (push 'g\_element 'l\_stack) ... 4-19  
 (pushnew 'g\_element 'l\_stack) ... 4-19  
 (putaccess 'a\_array 'su\_func) ... 2-29  
 (putaux 'a\_array 'g\_aux) ... 2-29  
 (putd 's\_name 'u\_func) ... 4-19  
 (putdata 'a\_array 'g\_arg) ... 2-29  
 (putdelta 'a\_array 'x\_delta) ... 2-29  
 (putdisc 'y\_func 's\_discipline) ... 2-31  
 (putlength 'a\_array 'x\_length) ... 2-29  
 (putprop 'ls\_name 'g\_val 'g\_ind) ... 2-33  
 q ... 15-2  
 (qualify-aexplode 's\_arg) ... 2-21  
 (qualify-aexplodec 's\_arg) ... 2-21  
 (qualify-aexploden 's\_arg) ... 2-21  
 (qualify-always-escape-aexploden 's\_arg) ... 2-21  
 (qualify-escape-aexploden 's\_arg) ... 2-21  
 (qualify-escape-exploden 'g\_arg) ... 2-20  
 (qualify-explode 'g\_arg) ... 2-20  
 (qualify-explodec 'g\_arg) ... 2-20  
 (qualify-exploden 'g\_arg) ... 2-20  
 (\*quo 'i\_x 'i\_y) ... 3-2  
 (quote g\_arg) ... 2-36  
 (quote! [g\_qform1] ... [! 'g\_iform1] ... [!! 'l\_form1] ...) ... 2-2  
 (quotient ['n\_arg1 ...]) ... 3-2  
 (r x y) ... 16-14  
 "~radix,mincolw,padchar:@r" ... 5-14  
 (random ['x\_limit]) ... 3-9  
 (rassq 'g\_arg1 'l\_arg2) ... 2-31  
 (ratom ['p\_port ['g\_eof]]) ... 5-8



(read ['p\_port ['g\_eof]]) ... 5-8  
 (readc ['p\_port ['g\_eof]]) ... 5-9  
 (readdir ['t\_dirname']) ... 5-9  
 (readline ['p\_port']) ... 5-9  
 (readlist 'l\_arg) ... 5-9  
 readtable ... B-4  
 "real-function" ... 18-5  
 (recompile-flavor 'flavor-name &optional 'single-op 'use-old-combined-methods 'do-  
     dependents) ... 19-22  
 REDO ... 15-6  
 REDO f ... 15-6  
 :rehash-size x\_rehash\_size ... 2-24  
 :rehash-threshold x\_rehash\_threshold ... 2-25  
 (remainder 'i\_dividend 'i\_divisor) ... 3-9  
 (rematom 's\_arg) ... 2-13  
 ?rem-function filen fcn1 [fcn2 ...] ... 13-13  
 (remhash 'g\_key 'H\_htab) ... 2-25  
 (remob 's\_symbol) ... 2-13  
 (remove 'g\_x 'l\_l) ... 2-6  
 (removeaddress 's\_name1 ['s\_name2 ...]) ... 5-9  
 (removeaddress 'st\_cfunc ...) ... 18-4  
 ?rem-prop filen (sym1 ind1) [(sym2 ind2) ...] ... 13-13  
 (remprop 'ls\_name 'g\_ind) ... 2-33  
 (remq 'g\_x 'l\_l ['x\_count']) ... 2-6  
 (remsym 'ls\_arg ...) ... 2-16  
 ?rem-var filen var1 [var2 ...] ... 13-13  
 (rename-package 'k\_pkname 's\_newname ['l\_newnicknames]) ... 17-6  
 repack ... 16-23  
 (repack \$) ... 16-23  
 (replace \$ with e1 ... em) ... 16-15  
 (replace 'g\_arg1 'g\_arg2) ... 2-36  
 (require 's\_name ['sl\_pathname]) ... 17-10  
 :required-flavors ... 19-26  
 :required-init-keywords defflavor ... 19-26  
 :required-instance-variables ... 19-26  
 :required-methods ... 19-26  
 (reset) ... 6-6  
 ?re[set] ... 13-3  
 (resetio) ... 5-9  
 ?ret [val] ... 13-4  
 "~RETURN" ... 5-16  
 RETURN ... 13-6, 15-1  
 RETURN e ... 15-6  
 (return ['g\_val]) ... 4-19



(reverse 'l\_arg) ... 2-8  
 (ri n m) ... 16-17  
 (ro n) ... 16-17  
 (rot 'x\_val 'x\_amt) ... 3-7  
 (rplaca 'lh\_arg1 'g\_arg2) ... 2-5  
 (rplacd 'lh\_arg1 'g\_arg2) ... 2-5  
 (rplacx 'x\_ind 'h\_hunk 'g\_val) ... 2-30  
 (\*rplacx 'x\_ind 'h\_hunk 'g\_val) ... 2-30  
 (\*rset 'g\_flag) ... 6-6  
 :run-time-alternatives defflavor ... 19-30  
 #S ... 8-10  
 #s ... 8-10  
 (s var) ... 16-23  
 (s var . \$) ... 16-23  
 (sassoc 'g\_arg1 'l\_arg2 'sl\_func) ... 2-32  
 (sassq 'g\_arg1 'l\_arg2 'sl\_func) ... 2-32  
 ?sc [n] ... 13-5  
 (scons 'x\_arg 'bs\_rest) ... 2-36  
 (second . \$) ... 16-13  
 (segment 's\_type 'x\_size) ... 6-7  
 (selectq 'g\_key-form [l\_clause1 ...]) ... 4-19  
 self ... 19-21  
 (send 's\_instance 's\_message ['argument ...]) ... 19-22  
 (send 's\_object 's\_message ['argument ...]) ... 19-8  
 :send-if-handles operation [arguments]... ... 19-33  
 (send-self 's\_message ['argument] ...) ... 19-22  
 "~sepwidth,linewidth;" ... 5-21  
 (set 's\_arg1 'g\_arg2) ... 2-19  
 (setarg 'x\_argnum 'g\_val) ... 4-19  
 (setf g\_accessfn1 'g\_val1) ... 14-2  
 (set-in-closure 'cl\_a 's\_symbol 'g\_x) ... 8-14  
 (set-in-fclosure 'v\_fclosure 's\_symbol 'g\_newvalue) ... 8-15  
 (set-in-instance 'instance 'symbol 'value) ... 19-23  
 (setplist 's\_atm 'l\_plist) ... 2-19, 2-33  
 (setq s\_atm1 'g\_val1 [ s\_atm2 'g\_val2 ... .. ]) ... 2-19  
 (setsyntax 's\_symbol 's\_synclass ['ls\_func]) ... 7-12  
 :settable-instance-variables ... 19-24  
 (shadow 'ls\_sym [k\_package]) ... 17-8  
 (shadowing-import 'lg\_sym ['k\_package]) ... 17-8  
 (shell) ... 6-7  
 (showstack) ... 6-7  
 (si:flavor-allowed-init-keywords 'flavor-name) ... 19-23  
 (signal 'x\_signum 's\_name) ... 6-7  
 (signp s\_test 'g\_val) ... 2-10



(sin 'fx\_angle) ... 3-5  
 (si:property-list-mixin :get property-name) ... 19-45  
 (si:property-list-mixin :getl property-name-list) ... 19-45  
 (si:property-list-mixin :property-list) ... 19-46  
 (si:property-list-mixin :push-property value property-name) ... 19-45  
 (si:property-list-mixin :putprop value property-name) ... 19-45  
 (si:property-list-mixin :remprop property-name) ... 19-45  
 (si:property-list-mixin :set-property-list list) ... 19-46  
 si:vanilla-flavor ... 19-33  
 :size x\_size ... 2-24  
 :size-macro name ... 14-9  
 (sizeof 'g\_arg) ... 6-7  
 :size-symbol name ... 14-9  
 (sload 's\_file) ... 5-9  
 (small-segment 's\_type 'x\_cells) ... 6-7  
 ?soff ... 13-5  
 (sort 'l\_data 'u\_comparefn) ... 2-37  
 (sortcar 'l\_list 'u\_comparefn) ... 2-37  
 ?sp [arg] ... 13-6  
 "~SPACE" ... 5-16  
 SPACE RETURN ... 13-6  
 :special-instance-variables ... 19-25  
 (sprintf 't\_control ['arg1 ...]) ... 5-9  
 (sqrt 'fx\_arg) ... 3-9  
 (sstatus appendmap g\_val) ... 6-8  
 (sstatus automatic-reset g\_val) ... 6-8  
 (sstatus chainatom g\_val) ... 6-8  
 (sstatus dumpcore g\_val) ... 6-8  
 (sstatus evalhook g\_val) ... 6-8  
 (sstatus feature g\_val) ... 6-8  
 (sstatus g\_type g\_val) ... 6-8  
 (sstatus nofeature g\_val) ... 6-9  
 (sstatus translink g\_val) ... 6-9  
 (sstatus uctolc g\_val) ... 6-9  
 ?state [sym1 val1 ...] ... 13-3  
 (status ctime) ... 6-9  
 (status feature g\_val) ... 6-9  
 (status features) ... 6-9  
 (status g\_code) ... 6-9  
 (status isatty) ... 6-10  
 (status localtime) ... 6-10  
 (status syntax s\_char) ... 6-10  
 (status undeffunc) ... 6-11  
 (status version) ... 6-11



**STEP ... 15-6**

**?step** [t | fn1 fn2 ...] ... 13-5

**(step** [t | nil | functor1 [functor2 ...]]) ... 15-1

**(sticky-bignum-leftshift** 'bx\_arg 'x\_amount) ... 3-5

**stop** ... 16-22

**(store** 'l\_arexp 'g\_val) ... 2-29

**(strcat** ['stn\_arg1 ...]) ... 2-12

**"~{string~:}"** ... 5-20

**"~{string~}"** ... 5-20

**"~:{string~:}"** ... 5-20

**"~:{string~}"** ... 5-20

**"~@{string~:}"** ... 5-20

**"~@{string~}"** ... 5-20

**"~:@{string~:}"** ... 5-20

**"~:@{string~}"** ... 5-20

**(string** 'st\_symbol-or-string) ... 2-19

**(string<** 't\_string1 't\_string2) ... 2-16

**(string=** 't\_string1 't\_string2) ... 2-16

**(string>** 't\_string1 't\_string2) ... 2-16

**(string/=** 't\_string1 't\_string2) ... 2-16

**(string<=** 't\_string1 't\_string2) ... 2-16

**(string>=** 't\_string1 't\_string2) ... 2-16

**"~<[string1~;string2...~;stringn~<]"** ... 5-18

**(string-capitalize** 't\_string) ... 2-17

**(string-downcase** 't\_string) ... 2-17

**(string-length** 't\_string) ... 2-17

**(stringp** 'g\_arg) ... 2-9

**(string-upcase** 't\_string) ... 2-17

**(sub1** 'n\_arg) ... 3-2

**(sublis** 'l\_alst 'l\_exp) ... 2-32

**(subpair** 'l\_old 'l\_new 'l\_expr) ... 2-7

**"subroutine"** ... 18-4

**(subst** 'g\_x 'g\_y 'l\_s) ... 2-7

**(substring** 'st\_string 'x\_index ['x\_length]) ... 2-18

**(substringn** 'st\_string 'x\_index ['x\_length]) ... 2-18

**(substringp** 't\_string1 't\_string2) ... 2-19

**(sum** ['n\_arg1 ...]) ... 3-1

**(sw n m)** ... 16-14

**(sxhash** 'g\_arg) ... 2-27

**(symbol-function** 's\_arg) ... 2-18

**(symbol-name** 's\_arg) ... 2-18

**(symbolp** 'g\_arg) ... 2-9

**(symbol-package** 's\_name) ... 2-18

**(symbol-plist** 's\_name) ... 2-33



(symbol-value 's\_arg) ... 2-17  
 (symeval 's\_arg) ... 2-17  
 (symeval-in-closure 'cl\_a 's\_x) ... 8-14  
 (symeval-in-fclosure 'v\_fclosure 's\_symbol) ... 8-15  
 (symeval-in-instance 'instance 'symbol ['no-error-p]) ... 19-23  
 (symstat 's\_arg ...) ... 2-16  
 (sys:access 'st\_filename 'x\_mode) ... 6-12  
 (sys:chmod 'st\_filename 'xl\_mode) ... 6-11  
 (sys:getpid) ... 6-12  
 (sys:getpwnam 'st\_name) ... 6-12  
 (sys:getpwuid 'x\_uid) ... 6-12  
 (sys:getuid) ... 6-12  
 (sys:link 'st\_oldfilename 'st\_newfilename) ... 6-12  
 (sys:time) ... 6-12  
 (sys:unlink 'st\_filename) ... 6-12  
 t ... B-5  
 (tab 'x\_col ['p\_port]) ... 5-10  
 "~<[*tag1*,*tag2*,...;*string1* *begin21*,*end21*,*begin22*,*end22*,...;*string2*~;;default" ... -  
 5-19  
 "~*tag1*,*tag2*,...;" ... 5-19  
 (tailp 'l\_x 'l\_y) ... 2-3  
 (tconc 'l\_ptr 'g\_x) ... 2-34  
 (terpr ['p\_port]) ... 5-10  
 (terpri ['p\_port]) ... 5-10  
 test ... 16-18  
 :test u\_compare ... 2-24  
 "~<*text1*~;;*text2*~;*text3*>" ... 5-21  
 (third . \$) ... 16-13  
 (throw 'g\_val [s\_tag]) ... 4-19  
 (\*throw 's\_tag 'g\_val) ... 4-19  
 (tilde-expand 'st\_name) ... 5-10  
 (times ['n\_arg1 ... ]) ... 3-2  
 (time-string ['x\_seconds]) ... 6-12  
 tl ... 16-23  
 TOP ... 15-5  
 top-level ... B-5  
 (top-level) ... 6-13  
 :top-level s\_user\_top\_level ... E-2  
 top-level-eofs ... 13-6  
 top-level-init ... 13-6  
 top-level-print ... 13-7  
 top-level-prompt ... 13-6  
 tpl-history-show ... 13-7  
 tpl-number-prompt ... 13-7



*tpl-prinlength* ... 13-7  
*tpl-prinlevel* ... 13-7  
*?tr* [fn1 fn2 ...] ... 13-3  
*(trace* [ls\_arg1 ...]) ... 11-1  
*(traceargs* s\_func [x\_level]) ... 11-4  
*(tracedump)* ... 11-5  
*trace-prinlength* ... 11-4  
*trace-prinlevel* ... 11-4  
*:tree* ... 14-13  
*"~@[true~<]"* ... 5-19  
*(truename* 'p\_port) ... 5-10  
*tty* ... 16-22  
*(tyl* ['p\_port]) ... 5-10  
*(typepeek* ['p\_port]) ... 5-10  
*(tyo* 'x\_char ['p\_port]) ... 5-10  
*(type* 'g\_arg) ... 2-10  
*:type* type-name ... 14-8  
*(typep* 'g\_arg) ... 2-10  
*(typep* 's\_item &optional 's\_flavor) ... 19-16  
*U* [n] [f] ... 15-6  
*(uconcat* ['stn\_arg1 ...]) ... 2-12  
*unblock* ... 16-18  
*(undefflavor* 'flavor) ... 19-21  
*(undefmethod* flavor [type] operation [suboperation]) ... 19-21  
*undo* ... 16-18  
*undo* ... 16-18  
*(unexport* 'ls\_sym ['k\_package]) ... 17-7  
*(unintern* 's\_symbol ['k\_package]) ... 17-7  
*(unless* pred form1 ...) ... 4-5  
*?untr* [fn1 fn2 ...] ... 13-3  
*(untrace* [s\_arg1 ...]) ... 11-5  
*(untyl* 'x\_char ['p\_port]) ... 5-10  
*(unuse-package* 'lkst\_packs [k\_package]) ... 17-8  
*(unwind-protect* g\_protected [g\_cleanup1 ...]) ... 4-19  
*up* ... 16-8  
*?up* [n] ... 13-4  
*UP* n ... 15-6  
*(use-package* 'lkst\_packs [k\_package]) ... 17-8  
*usermacros* ... 16-22  
*user-top-level* ... B-5, 13-6  
*(valuep* 'g\_arg) ... 2-9  
*(values* ['g\_arg1 ... 'g\_argn]) ... 4-20  
*(values-llst* 'l\_arg) ... 4-20  
*vcharacter* ... 7-7



**:vector** ... 14-14  
**(vector** [*'g\_val0 'g\_val1 ...*]) ... 2-21  
*"vector-c-function"* ... 18-5  
**(vectori-byte** [*'x\_val0 'x\_val1 ...*]) ... 2-22  
**(vectori-byte-to-string** *'Vv\_vect*) ... 2-23  
**(vectori-double** [*'f\_val0 'f\_val1 ...*]) ... 2-22  
**(vectori-float** [*'f\_val0 'f\_val1 ...*]) ... 2-22  
**(vectori-long** [*'x\_val0 'x\_val1 ...*]) ... 2-22  
**(vectorip** *'v\_vector*) ... 2-9  
**(vectori-word** [*'x\_val0 'x\_val1 ...*]) ... 2-22  
**(vectorp** *'g\_arg*) ... 2-22  
**(vectorp** *'v\_vector*) ... 2-9  
*vescape* ... 7-8  
**(vget** *'Vv\_vect 'g\_ind*) ... 2-22  
*villegal* ... 7-10  
*vinfix-macro* ... 7-9  
*vleft-bracket* ... 7-7  
*vleft-paren* ... 7-7  
*vmacro* ... 7-9  
*vnumber* ... 7-7  
*"void-c-function"* ... 18-5  
*vpackage* ... 7-7  
*vperiod* ... 7-8  
**(vprop** *'Vv\_vect*) ... 2-22  
**(vputprop** *'Vv\_vect 'g\_value 'g\_ind*) ... 2-23  
*"~v:@q"* ... 5-17  
**(vref** *'v\_vect 'x\_index*) ... 2-22  
**(vrefi-byte** *'V\_vect 'x\_bindex*) ... 2-22  
**(vrefi-double** *'V\_vect 'x\_lindex*) ... 2-22  
**(vrefi-float** *'V\_vect 'x\_lindex*) ... 2-22  
**(vrefi-long** *'V\_vect 'x\_lindex*) ... 2-22  
**(vrefi-word** *'V\_vect 'x\_windex*) ... 2-22  
*vright-bracket* ... 7-7  
*vright-paren* ... 7-7  
*vseparator* ... 7-8  
**(vset** *'v\_vect 'x\_index 'g\_val*) ... 2-23  
**(vseti-byte** *'V\_vect 'x\_bindex 'x\_val*) ... 2-23  
**(vseti-double** *'V\_vect 'x\_lindex 'f\_val*) ... 2-23  
**(vseti-float** *'V\_vect 'x\_lindex 'f\_val*) ... 2-23  
**(vseti-long** *'V\_vect 'x\_lindex 'x\_val*) ... 2-23  
**(vseti-word** *'V\_vect 'x\_windex 'x\_val*) ... 2-23  
**(vsetprop** *'Vv\_vect 'g\_value*) ... 2-23  
*vsign* ... 7-7  
*vsingle-character-symbol* ... 7-9



*vsingle-infix-macro* ... 7-10  
*vsingle-macro* ... 7-9  
*vsingle-quote* ... 7-8  
*vsingle-splicing-macro* ... 7-9  
(*vsized* 'Vv\_vect) ... 2-22  
(*vsized-byte* 'V\_vect) ... 2-23  
(*vsized-double* 'V\_vect) ... 2-23  
(*vsized-float* 'V\_vect) ... 2-23  
(*vsized-word* 'V\_vect) ... 2-23  
*vsplicing-macro* ... 7-9  
*vstring-delimiter* ... 7-8  
*vsymbol-delimiter* ... 7-8  
*^w* ... B-2  
(*wait*) ... 6-13  
(*when pred form1 ...*) ... 4-5  
**WHERE** ... 15-6  
*?whichfile* fcn | var | (symbol ind) ... 13-13  
**:which-operations** ... 19-33  
(*wide-print-llist* 'g\_exp [keyword value]) ... 5-11  
*"~widtha"* ... 5-13  
*"~widths"* ... 5-14  
**:wrapper** ... 19-41  
**#X** ... 8-11  
**#x** ... 8-11  
(*xcons* 'g\_arg1 'g\_arg2) ... 2-1  
(*xtr* . \$) ... 16-15  
(*y-or-n-p* ['t\_message]) ... 5-11  
(*zapline*) ... 5-11  
(*zerop* 'g\_arg) ... 3-3  
*?zo* ... 13-4



## B Special symbols







## B Special symbols

The values of these symbols have a predefined meaning. Some values are counters, while others are simply flags whose value the user can change to affect the operation of the LISP system. In all cases, only the value cell of the symbol is important; the function cell is not. The value of some of the symbols (like *ER%misc*) are functions. What this means is that the value cell of those symbols either contains a *lambda* expression, a binary object, or symbol with a function binding.

The values of the special symbols are:

*\$gccount\$*

- The number of garbage collections which have occurred.

*\$gcprint*

- If bound to a non-*nil* value, then, after each garbage collection and subsequent storage allocation, a summary of storage allocation is printed.

*\$ldprint*

- If bound to a non-*nil* value, then, during each **fasl** or **cfasl**, a diagnostic message is printed.

*ER%all*

- The function that is the error handler for all errors. (See Chapter 10).

*ER%brk*

- The function that is the handler for the error signal generated by the evaluation of the **break** function. (See Chapter 10).

*ER%err*

- The function that is the handler for the error signal generated by the evaluation of the **err** function. (See Chapter 10).



*ER%misc*

- The function that is the handler of the error signal generated by one of the unclassified errors. (See Chapter 10). Most errors are unclassified at this point.

*ER%tpl*

- The function that is the handler to be called when an error has occurred which has not been handled. (See Chapter 10).

*ER%undef*

- The function that is the handler for the error signal generated when a call to an undefined function is made.

*^w*

- When it is bound to a non-*nil* value, this prevents output to the standard output port (*oport*) from reaching the standard output (usually a terminal). Note that *^w* is a two character symbol and should not be confused with *#^W* which is how the ASCII character *control-W* is denoted. The value of *^w* is checked when the standard output buffer is flushed, which occurs after a *terpr*, *drain*, or when the buffer overflows. This is most useful in conjunction with *oport* described later. System error handlers rebind *^w* to *nil* when they are invoked to ensure that error messages are not lost. (This was introduced for MacLISP compatibility.)

*defmacro-for-compiling*

- This has an effect during compilation. If it is non-*nil*, it causes macros defined by *defmacro* to be compiled and included in the object file.

*displace-macros*

- If this symbol is non-*nil*, *eval* replaces calls to macros with the expanded form of the macros.

*environment*

- The operating system environment in *assoc* list form.

*errlist*

- When a *reset* is done, the value of *errlist* is saved away and control is thrown to the top level. *eval* is then mapped over the saved away value of this list.



*errport*

- This port is initially bound to the standard error file.

*evalhook*

- The value of this symbol, if bound, is the name of a function to handle *evalhook* traps (see §15.4).

*float-format*

- The value of this symbol is a string that is the format to be used by *print* to print *flonums*. See the documentation on the operating system function *printf* for a list of allowable formats.

*funcallhook*

- The value of this symbol, if bound, is the name of a function to handle *funcallhook* traps. (See Chapter §15.4.)

*gcdisable*

- If it is non-*nil*, then garbage collections are not done automatically when a collectable data type runs out.

*ibase*

- This is the input radix used by the LISP reader. It may be either eight or ten. Numbers followed by a decimal point are assumed to be decimal regardless of what *ibase* is.

*lisp-library-directory*

- The value of this symbol is the name of the directory containing the LISP library.

*lisp-object-directory*

- The value of this symbol is the name of the directory containing the LISP binary.

*linel*

- The line length used by the pretty printer, *pp*. This should be used by *print* but it is not at this time.

*load-most-recent*

- If set to *t*, *load* (and the *?ld* command) will load the the interpreted source file (the *.l* file) instead of the compiled ( *.o* file) if it is newer, and the argument to the *load* command does not contain either a *.l* or *.o* file extension.



*multiple-values-limit*

- The maximum number of multiple values that can be returned. This is a read-only variable.

*nil*

- This symbol represents the null list and, thus, can be written *()* or *'()* or *'nil*. Its value is always *nil*. Any attempt to change the value results in an error.

*\*package\**

- The value of this symbol is the current package. See chapter 17.

*piport*

- Initially bound to the standard input (usually the keyboard). A read with no arguments reads from *piport*.

*poport*

- Initially bound to the standard output (usually the terminal console). A print with no second argument writes to *poport*. See also: *^w* and *ptport*.

*prinlength*

- If this is a positive *fixnum*, then the *print* function prints no more than *prinlength* elements of a list or hunk and further elements abbreviated as *". . ."*. The initial value of *prinlength* is *nil*.

*prinlevel*

- If this is a positive *fixnum*, then the *print* function prints only *prinlevel* levels of nested lists or hunks. Lists below this level are abbreviated by *"&"* and hunks below this level are abbreviated by a *%*. The initial value of *prinlevel* is *nil*.

*ptport*

- Initially bound to *nil*. If bound to a port, then all output sent to the standard output is also sent to this port as long as this port is not also the standard output since this would cause a loop. Note that *ptport* does not get a copy of whatever is sent to *poport* if *poport* is not bound to the standard output.

*readtable*

- The value of this is the current *readtable*. It is an array, but you should NOT try to change the value of the elements of the array using the array functions. This is because the *readtable* is an array of bytes and the smallest unit the array functions work with is a full word (4 bytes). You can use *setsyntax* to change the values and



(status syntax ...) to read the values.

*t*

■ This symbol always has the value *t*. It is possible to change the value of this symbol for short periods of time, but you are strongly advised against it.

#### *top-level*

■ In a LISP system without */usr/lib/lisp/tpl.l* loaded, after a **reset** is done, the LISP system **funcalls** the value of *top-level* if it is non-*nil*. This provides a way for you to introduce your own top-level interpreter. When */usr/lib/lisp/tpl.l* is loaded, it sets *top-level* to **tpl** and changes the **reset** function so that once **tpl** starts, it cannot be replaced by changing *top-level*. **tpl** does provide a way of changing the top level however, and that is through *user-top-level*.

#### *user-top-level*

■ If this is bound, then after a **reset** the top level function **funcalls** the value of this symbol rather than going through a read eval print loop.



1. The first part of the report deals with the general situation of the country and the position of the various groups.

2. The second part of the report deals with the economic situation and the measures taken to improve it.

3. The third part of the report deals with the social situation and the measures taken to improve it.



## C The garbage collector







## C The garbage collector

The FRANZ LISP storage management *garbage collector* is invoked automatically whenever a collectable data type's current allocation is exhausted. All data types are collectable except for strings. After a garbage collection finishes, the collector calls the function **gcafter**, which should be a *lambda* of one argument. The argument passed to **gcafter** is the name of the data type that ran out and which caused the garbage collection. It is **gcafter**'s responsibility to allocate more pages of free space. The default **gcafter** makes its decision based on the percentage of space still in use after the garbage collection. If there is a large percentage of space still in use, **gcafter** allocates a larger amount of free space than if only a small percentage of space is still in use. The default **gcafter** also prints a summary of the space in use if the variable *\$gcprint* is non-*nil*. The summary always includes the state of the list and *fixnum* space, and includes an additional type if that type caused the garbage collection. The type that provoked the garbage collection is preceded by an asterisk.

15-05-1972

The first of the three main types of  
the second of the three main types of  
the third of the three main types of  
the fourth of the three main types of  
the fifth of the three main types of  
the sixth of the three main types of  
the seventh of the three main types of  
the eighth of the three main types of  
the ninth of the three main types of  
the tenth of the three main types of  
the eleventh of the three main types of  
the twelfth of the three main types of  
the thirteenth of the three main types of  
the fourteenth of the three main types of  
the fifteenth of the three main types of  
the sixteenth of the three main types of  
the seventeenth of the three main types of  
the eighteenth of the three main types of  
the nineteenth of the three main types of  
the twentieth of the three main types of  
the twenty-first of the three main types of  
the twenty-second of the three main types of  
the twenty-third of the three main types of  
the twenty-fourth of the three main types of  
the twenty-fifth of the three main types of  
the twenty-sixth of the three main types of  
the twenty-seventh of the three main types of  
the twenty-eighth of the three main types of  
the twenty-ninth of the three main types of  
the thirtieth of the three main types of  
the thirty-first of the three main types of  
the thirty-second of the three main types of  
the thirty-third of the three main types of  
the thirty-fourth of the three main types of  
the thirty-fifth of the three main types of  
the thirty-sixth of the three main types of  
the thirty-seventh of the three main types of  
the thirty-eighth of the three main types of  
the thirty-ninth of the three main types of  
the fortieth of the three main types of  
the forty-first of the three main types of  
the forty-second of the three main types of  
the forty-third of the three main types of  
the forty-fourth of the three main types of  
the forty-fifth of the three main types of  
the forty-sixth of the three main types of  
the forty-seventh of the three main types of  
the forty-eighth of the three main types of  
the forty-ninth of the three main types of  
the fiftieth of the three main types of  
the fifty-first of the three main types of  
the fifty-second of the three main types of  
the fifty-third of the three main types of  
the fifty-fourth of the three main types of  
the fifty-fifth of the three main types of  
the fifty-sixth of the three main types of  
the fifty-seventh of the three main types of  
the fifty-eighth of the three main types of  
the fifty-ninth of the three main types of  
the sixtieth of the three main types of  
the sixty-first of the three main types of  
the sixty-second of the three main types of  
the sixty-third of the three main types of  
the sixty-fourth of the three main types of  
the sixty-fifth of the three main types of  
the sixty-sixth of the three main types of  
the sixty-seventh of the three main types of  
the sixty-eighth of the three main types of  
the sixty-ninth of the three main types of  
the seventieth of the three main types of  
the seventy-first of the three main types of  
the seventy-second of the three main types of  
the seventy-third of the three main types of  
the seventy-fourth of the three main types of  
the seventy-fifth of the three main types of  
the seventy-sixth of the three main types of  
the seventy-seventh of the three main types of  
the seventy-eighth of the three main types of  
the seventy-ninth of the three main types of  
the eightieth of the three main types of  
the eighty-first of the three main types of  
the eighty-second of the three main types of  
the eighty-third of the three main types of  
the eighty-fourth of the three main types of  
the eighty-fifth of the three main types of  
the eighty-sixth of the three main types of  
the eighty-seventh of the three main types of  
the eighty-eighth of the three main types of  
the eighty-ninth of the three main types of  
the ninetieth of the three main types of  
the ninety-first of the three main types of  
the ninety-second of the three main types of  
the ninety-third of the three main types of  
the ninety-fourth of the three main types of  
the ninety-fifth of the three main types of  
the ninety-sixth of the three main types of  
the ninety-seventh of the three main types of  
the ninety-eighth of the three main types of  
the ninety-ninth of the three main types of  
the hundredth of the three main types of



D Lxref





## D Lxref

**lxref** reads cross-reference files written by the LISP compiler **liszt** and prints a cross-reference listing on the standard output. **Liszt** will create a cross reference file during compilation when it is given the **-x** switch. Cross-reference files usually end in **.x** and consequently **lxref** will append a **.x** to the file names given if necessary.

The **lxref** command line looks like:

```
% lxref [-N] xref-file ... [-a source-file ...] [-A source-file]
```

The first option to **lxref** is a decimal integer, **N**, which sets the **ignorelevel**. If a function is called more than **ignorelevel** times, the cross-reference listing will just print the number of calls instead of listing each one of them. The default for **ignorelevel** is 50.

The **-a** option causes **lxref** to put limited cross reference information in the sources named. **lxref** will scan the source and when it comes across a definition of a function (that is a line beginning with **(def** it will precede that line with a list of the functions which call this function, written as a comment preceded by **;**... All existing lines beginning with **;**.. will be removed from the file. If the source file contains a line beginning **;-** then this will disable this annotation process from this point on until a **;**+ is seen (however, lines beginning with **;**.. will continue to be deleted). After the annotation is done, the original file **foo.l** is renamed to **#foo.l** and the new file with annotation is named **foo.l**.

The **-A** switch, is like **-a** except it associates up to two function names per function. For example if you have something **(def-md s-foo md-foo (x)** then **lxref** will write in the file the names of the functions that call either **s-foo** or **md-foo**.

1901

The first of the year was a very dry one, and the crops were much injured. The weather was very hot, and the crops were much injured. The weather was very hot, and the crops were much injured.

The second of the year was a very wet one, and the crops were much injured. The weather was very cold, and the crops were much injured. The weather was very cold, and the crops were much injured.

The third of the year was a very dry one, and the crops were much injured. The weather was very hot, and the crops were much injured. The weather was very hot, and the crops were much injured.



# E Reconfiguring lisp

## 1.1 Changing the library directory E-2





## E Reconfiguring lisp

The file *franz/h/m-machine.h* contains definitions of the internal parameters for LISP. In order to modify this file you should have a familiarity with the 'C' programming language.

A few of these parameters can be changed, others have values determined by the machine and operating system. Some of the parameters are defined on a per-machine basis, e.g. those for the Tektronix 4404 are surrounded by `#ifdef pegasus ... #endif`, and those for the SUN are surrounded by `#ifdef sunXX ... #endif`.

The important parameters that you may wish to alter are:

**#define TTSIZE 6120**

■ The value is the maximum size that lisp can grow. It is measured in 512-byte *pages*, thus the default value of 6120 means that the maximum size is slightly more than 3 megabytes. (On some machines, the default value is different.) There is an 18 byte overhead in static table space for each potential LISP page.

□ Note that this parameter is defined in a machine-specific 'C' include file *franz/h/m-machine.h*.

**#define NAMESIZE 3072**

**#define NAMINC 50**

■ **NAMESIZE** defines the number of entries in the 'namestack' which is the stack used for holding function arguments and local (non-special) variables. It is rare that a correctly functioning program will exceed the default size of 3072 entries, so if you get a namestack overflow, make sure that your program isn't in an infinite loop before changing this value and rebuilding lisp. The size allocated is **NAMESIZE + 4 \* NAMINC**, the stack growing by **NAMINC** each time it overflows.

**#define XstackSize**

■ The number of entries in the alternate stack. This stack is used for *bignum* arithmetic and *fasl*. If you get an *out of alternate stack* message, you can increase this parameter.

To rebuild LISP, change to the directory *franz/machine* where *machine* is *68k* for machines based on the Motorola 68000 series microprocessors, *32k* for machines based on the National Semiconductor 32000 series microprocessors, *vax* for DEC VAX machines running Unix, *is25* for AT&T 3B series machines (except for the 3B1 which is



a 68k machine), *eunice* for DEC VAX machines running Eunice, and *vms* for DEC VAX machines running VMS. Then type **update updatefile** on the Tek4404, and **make -f Make.user** on all others. The result will be a new version of LISP named *nlisp*.

## 1.1 Changing the library directory

(**si:change-library-directory** [keyword value ... ...])

- **SIDE EFFECT:** Dumps a new executable binary of the current LISP session.

**:libdir** *t\_libdir*

- The *t\_libdir* argument is the path name to the directory containing the LISP library. FRANZ LISP uses *t\_libdir* to determine where to look for files that may need to be autoloaded.

**:objdir** *t\_objdir*

- The *t\_objdir* argument is the path name to the directory containing the executable binaries for FRANZ LISP. This information is needed by LISP for the *autorun* facility.

**:dumpname** *s\_dumpname*

- *dumpname* is the name of the new executable image that will be created by **change-library-directory**.

**:top-level** *s\_user\_top\_level*

- The value of *user-top-level* is set to *s\_user\_top\_level* in the newly created binary executable image.



# Index





# Index

#\ character macro 8-10  
 #- character macro 8-10  
 #+ character macro 8-10  
 #, character macro 8-10  
 #. character macro 8-10  
 #: character macro 8-10  
 #| character macro 8-10  
 |# character macro 8-10  
 #+ character macro 8-11  
 #/ character macro 8-11  
 #^ character macro 8-11  
 #- character macro 8-12  
 #' character macro 8-9  
 #( character macro 8-9  
 ? fixit command 15-6  
 "% format directive 5-15  
 "& format directive 5-16  
 "]" format directive 5-16  
 "-" format directive 5-16  
 "\*" format directive 5-18  
 ":[ format directive 5-18  
 ";;" format directive 5-18  
 "]" format directive 5-18  
 "]" format directive 5-18  
 ";;" format directive 5-19  
 ";;" format directive 5-19  
 "<[;" format directive 5-19  
 "@[" format directive 5-19  
 "@[<]" format directive 5-19  
 ":[{:] format directive 5-20  
 ":[{}]" format directive 5-20  
 ":[{}]" format directive 5-20  
 ":[{}]" format directive 5-20  
 "@[{}]" format directive 5-20  
 "@[{}]" format directive 5-20  
 "{ format directive 5-20  
 "{:}" format directive 5-20  
 "{ }" format directive 5-20  
 "]" format directive 5-20

";;" format directive 5-21  
 "<" format directive 5-21  
 ">" format directive 5-21  
 "" format directive 5-22  
 (\#) function 16-6  
 (+) function 3-1  
 (-) function 3-2  
 (\*) function 3-2  
 (/) function 3-2  
 (<&) function 3-4  
 (<) function 3-4  
 (<=&) function 3-4  
 (= &) function 3-4  
 (=) function 3-4  
 (>&) function 3-4  
 (>) function 3-4  
 (>=&) function 3-4  
 \$ lisp editor command 16-12  
 ?? lisp editor command 16-18  
 / lisp editor command 16-7  
 \_ lisp editor command 16-7  
 \_<\_ lisp editor command 16-7  
 ? lisp editor command 16-8  
 !! lisp editor command 16-9  
 ^ lisp editor command 16-9  
 (!) lisp editor form 16-14  
 ( ) lisp editor form 16-7  
 ?? tpl command 13-2  
 0 lisp editor command 16-8  
 !0 lisp editor command 16-9  
 (1+) function 3-1  
 (1-) function 3-2  
 (\$1) lisp editor form 16-17, 16-18



**"~a"** format directive 5-13  
**(a)** lisp editor form 16-14  
 A simple example (§18.1) 18-1  
 A top-level for debugging LISP programs (§13.2) 13-1  
**(abs)** function 3-8  
**:abstract-flavor** flavor keyword option 19-30  
**(absval)** function 3-8  
 Access function (§9.2.1) 9-2  
 Accessing C structures from LISP (§18.4.6) 18-11  
**:accessor-prefix** flavor keyword option 19-29  
**(acos)** function 3-5  
**(add)** function 3-1  
**(add1)** function 3-1  
**?add-function** tpl command 13-13  
**(addhash)** function 2-25  
**?add-prop** tpl command 13-13  
**(add-syntax-class)** function 7-13  
**?add-var** tpl command 13-13  
 Advanced features (§15.2) 15-2  
**(aexplode)** function 2-21  
**(aexplodec)** function 2-21  
**(aexploden)** function 2-21  
**:after** flavor keyword option 19-40  
**:alias-flavor** flavor keyword option 19-29  
**\*all-flavor-names\*** variable 19-16  
**(allocate)** function 6-1  
**(allsym)** function 2-16  
**(alphalessp)** function 2-16  
**:alterant** defstruct keyword 14-10  
 Alterant macros (§14.3.1.1.3) 14-15  
**(alter-name)** function 14-15  
 Alternate strings (§5.3.3.1) 5-18  
 An example (§18.4.6.1) 18-11 (§8.4.1) 8-13  
 Anatomy of vectors (§9.5) 9-4  
**:and** flavor keyword option 19-38, 19-40, 19-41  
**(and)** function 4-1  
**:append** flavor keyword option 19-38, 19-41  
**(append)** function 2-1  
**(append1)** function 2-2  
**(apply)** function 4-1  
**(apropos)** function 4-2

**(apropos-list)** function 4-2  
**(arg)** function 4-3  
**(argv)** function 6-1  
 Arithmetic and logic functions (Chapter 3) 3-1  
 Array (§1.1.11) 1-4  
**:array** defstruct type 14-13  
**(array)** function 2-27  
**(\*array)** function 2-27  
 Array accessors (§18.4.6.4) 18-15 (§2.6.3) 2-28  
 Array creation (§2.6.1) 2-27  
 Array manipulation (§2.6.4) 2-29  
 Array predicate (§2.6.2) 2-28  
**(arraycall)** function 2-28  
**(arraydims)** function 2-28  
**(arrayp)** function 2-27, 2-8  
**(arrayref)** function 2-28  
 Arrays (§2.6) 2-27  
 Arrays and vectors (Chapter 9) 9-1  
**(ascii)** function 2-13  
**(ash)** function 3-6  
**(asin)** function 3-5  
**(assoc)** function 2-31  
 Association lists (§2.9.1) 2-31  
**(assq)** function 2-31  
**(atan)** function 3-5  
**(atom)** function 2-8  
**(attach)** function 2-5  
 Autoloading (§10.5) 10-3  
 Autorun (§12.6) 12-7  
 Auxiliary (§9.2.2) 9-2  
  
**(b)** lisp editor form 16-14  
**b** step command 15-2  
 Backup variables in the file package (§13.6.2) 13-14  
**(baktrace)** function 6-1  
**(bcdad)** function 2-35  
**(bcdp)** function 2-8  
 Bcds (§2.8) 2-31  
**:before** flavor keyword option 19-40  
**(below)** lisp editor form 16-10  
**bf** lisp editor command 16-12  
**(bf)** lisp editor form 16-12  
**(bi)** lisp editor form 16-16  
 Bignum (§1.1.7) 1-3



- Bignum and fixnum manipulation (§3.4) 3-5
- (**bignum-leftshift**) function 3-5
- (**bignum-to-list**) function 2-3
- (**bigp**) function 2-8
- Binary (§1.1.4) 1-2
- (**bind**) lisp editor form 16-21
- Bit manipulation (§3.5) 3-6
- bk** lisp editor command 16-9
- BKA** fixit BK command suffix 15-7
- BKC** fixit BK command suffix 15-7
- BKE** fixit BK command suffix 15-7
- BKF** fixit BK command suffix 15-7
- BKsuffix;** fixit command 15-7
- BKV** fixit BK command suffix 15-7
- (**bo**) lisp editor form 16-16
- (**boole**) function 3-6
- BOT** fixit command 15-5
- (**boundp**) function 2-16
- :break** flavor keyword option 19-34
- (**break**) function 4-3
- (**\*break**) function 4-3
- Built-in packages (§17.5) 17-4
- :but-first** defstruct keyword 14-10
  
- "**~c**" format directive 5-15
- c** step command 15-1
- ?c** tpl stepper command 13-5
- :callable-accessors** defstruct keyword 14-9
- Call-by-address disciplines (§18.3.1) 18-4
- Call-by-value disciplines (§18.3.2) 18-5
- Calling C functions (§18.4) 18-5
- Calling LISP from C (§18.5.3) 18-17
- (**car**) function 2-3
- :case** flavor keyword option 19-39
- (**case**) function 4-3
- (**caseq**) function 4-3
- (**catch**) function 4-4
- (**\*catch**) function 4-4
- ccharacter** character class 7-2
- (**c-declare**) macro 18-12
- (**cdr**) function 2-3
- (**cerror**) function 4-11
- cescape** character class 7-5
- (**cfasl**) function 18-3, 5-1
- "**c-function**" function discipline 18-5
- (**change**) lisp editor form 16-15
- ?changed** tpl command 13-13
- (**change-library-directory**) function E-2
- Changing a flavor (§19.13.2) 19-44
- Changing the library directory (§1.1) E-2
- Character classes (§7.4) 7-3
- Character macros (§7.6) 7-10
- (**charcnt**) function 5-2
- (**char-index**) function 2-19
- (**char-rindex**) function 2-19
- (**chdir**) function 6-2
- cillegal** character class 7-6
- cinfix-macro** character class 7-6
- Classification (§12.3.2) 12-2
- cleft-bracket** character class 7-3
- cleft-paren** character class 7-3
- (**cli:error**) function 4-11
- (**close**) function 5-2
- Closure (§1.1.15) 1-5
- (**closure**) function 8-13
- (**closurep**) function 2-9
- Closures and fclosures (§8.4) 8-12
- (**clrhash**) function 2-25
- cmacro** character class 7-5
- cnumber** character class 7-3
- :combined** flavor keyword option 19-41
- (**command-line-args**) function 6-2
- Commands in the file package (§13.6.1) 13-12
- Commands that evaluate (§16.11) 16-19
- Commands that test (§16.12) 16-20
- (**comment**) function 4-4
- (**compile-flavor-methods**) function 19-22
- Compiler options (§12.5) 12-5
- (**coms**) lisp editor form 16-19
- (**comsq**) lisp editor form 16-19
- (**concat**) function 2-12
- (**concatl**) function 2-12
- :conc-name** defstruct keyword 14-5
- (**cond**) function 4-5
- Conditional and iterative directives (§5.3.3) 5-18
- Conditional inclusion (§8.3.5.1) 8-11
- (**cons**) function 2-1
- Consistency rules (§17.1.1) 17-2
- :constructor** defstruct keyword 14-8
- Constructor functions (§14.3.1.1.2) 14-14
- (**copy**) function 2-35
- lisp editor form 16-16



- Copying instances (§19.15) 19-46
- (**copyint\***) function 2-35
- (**copy-symbol**) function 2-13
- (**copysymbol**) function 2-13
- (**cos**) function 3-5
- (**cp**) lisp editor form 16-16
- cpackage** character class 7-3
- cperiod** character class 7-4
- (**cprintf**) function 5-2
- (**cpy1**) function 2-35
- (**c..r**) function 2-4
- Creating packages in files (§17.8) 17-10
- cright-bracket** character class 7-4
- cright-paren** character class 7-3
- cseparator** character class 7-4
- csign** character class 7-3
- csingle-character-symbol** character class 7-5
- csingle-infix-macro** character class 7-6
- csingle-macro** character class 7-5
- csingle-quote** character class 7-4
- csingle-splicing-macro** character class 7-6
- csplicing-macro** character class 7-5
- cstring-delimiter** character class 7-5
- csymbol-delimiter** character class 7-4
- (**cvtttofranzlisp**) function 4-5
- (**cvttointlisp**) function 4-5
- (**cvttomacilisp**) function 4-5
- (**cvttoucillisp**) function 4-5
- (**cxr**) function 2-30
  
- "**~d**" format directive 5-14
- d** step command 15-2
- :daemon** flavor keyword option 19-36
- :daemon-with-and** flavor keyword option 19-37
- :daemon-with-or** flavor keyword option 19-36
- :daemon-with-override** flavor keyword option 19-37
- Data (§9.2.3) 9-2
- Data structure access (Chapter 2) 2-1
- Data types (§1.1) 1-1
- (**debug**) function 15-5, 4-6
- ?**debug** tpl command 13-2
- ?**debug full** tpl command 13-2
- (**debugging**) function 4-6
- Debugging tools (§18.4.6.7) 18-16
  
- Declare (§12.3.2.2) 12-2
- (**declare**) function 4-6
- Def (§12.3.2.5) 12-4
- (**def**) function 4-6
- :default** flavor keyword option 19-40
- Default aids (§10.4) 10-3
- :default-handler** flavor keyword option 19-28
- :default-init-plist** flavor keyword option 19-25
- :default-pointer** defstruct keyword 14-8
- (**defcmacro**) function 4-6
- (**defflavor**) function 19-16
- Def flavor options (§19.9) 19-24
- #define NAMESIZE** token E-1
- #define NAMINC** token E-1
- #define TTSIZE** token E-1
- #define XstackSize** token E-1
- Defmacro (§8.3.2) 8-7
- (**defmacro**) function 4-6
- defmacro-for-compiling** constant B-2
- (**defmethod**) function 19-17
- (**defprop**) function 2-33
- (**defsetf**) function 14-2
- (**defsharp**) macro 8-9
- Defstruct (§14.3) 14-4
- (**defstruct**) function 14-5
- Defstruct argument forms (§14.3.1) 14-5
- (**defsubst**) function 4-6
- (**defun**) function 4-7
- (**defvar**) function 4-8
- (**defwhopper**) function 19-21
- (**defwrapper**) function 19-20
- (**delete**) function 2-5
- delete** lisp editor command 16-14
- (**delete**) lisp editor form 16-14
- (**delq**) function 2-6
- Delta (§9.2.5) 9-3
- (**deref**) function 6-2
- :describe** flavor keyword option 19-33
- (**describe-cs**) macro 18-16
- (**describe-flavor**) function 19-23
- (**desetq**) function 2-19
- (**diff**) function 3-2
- (**difference**) function 3-2
- displace-macros** constant B-2
- tpl special symbol 13-7
- (**Divide**) function 3-2



- DN fixit command 15-6
- ?dn tpl command 13-4
- DNFN fixit command 15-6
- (do) function 4-8, 4-9
- (do\*) function 4-9
- (do-all-symbols) function 17-9
- :documentation flavor keyword option 19-32
- Documentation conventions (§1.2) 1-6
- (do-external-symbols) function 17-9
- (dolist) function 4-9
- (do-symbols) function 17-9
- (dotimes) function 4-9
- "double-c-function" function discipline 18-5
- (double-to-float) function 18-16
- (drain) function 5-3
- (dremove) function 2-6
- (dsubst) function 2-7
- (dtptr) function 2-3, 2-9
- (dumplisp) function 6-2
- :dumpname change-library-directory keyword E-2
- "~e" format directive 5-14
- e lisp editor command 16-19
- (e) lisp editor form 16-19
- e step command 15-2
- Edit chain command summary (§16.3.1.1) 16-7
- editcomsl variable 16-22
- (editf) function 16-5
- (editfindp) function 16-6
- (editfns) function 16-6
- Editor functions (§16.3) 16-5
- Editor macros (§16.13) 16-20
- (editp) function 16-5
- (editracefn) function 16-6
- (editv) function 16-5
- (embed) lisp editor form 16-15
- (Emuldiv) function 3-3
- environment constant B-2
- (environment) function 4-10
- (environment-lmlisp) function 4-10
- (environment-maclisp) function 4-10
- EOF tpl command 13-4
- (eq) function 2-10
- (eq!) function 2-11
- (eqstr) function 2-11
- (equal) function 2-11
- ER%all constant B-1
- ER%brk constant B-1
- ER%err constant B-1
- ER%misc constant B-2
- (err) function 4-11
- errlist constant B-2
- (error) function 4-11
- Error handling algorithm (§10.3) 10-2
- errport constant B-3
- (errset) function 4-11
- Errset and error handler functions (§10.1) 10-1
- ER%tpl constant B-2
- ER%undef constant B-2
- (escape-aexploden) function 2-21
- (escape-exploden) function 2-20
- ?ev tpl command 13-4
- (eval) function 4-12
- (evalframe) function 4-12
- evalhook constant B-3
- (evalhook) function 4-13
- Evalhook and funcallhook (§15.4) 15-3
- :eval-inside-yourself flavor keyword option 19-34
- Eval-when (§12.3.2.1) 12-2
- :eval-when defstruct keyword 14-10
- (eval-when) function 6-2
- (evenp) function 3-3
- Exception handling (Chapter 10) 10-1
- (exec) function 6-2
- (exece) function 6-3
- (exit) function 6-3
- (exp) function 3-8
- (explode) function 2-20
- (explodec) function 2-20
- (exploden) function 2-20
- (export) function 17-7
- Exporting and importing symbols (§17.4) 17-3
- \*expr, compiler declaration 12-2
- (expt) function 3-8
- External references (§18.5.1) 18-16
- (extract) lisp editor form 16-15
- Extraction and embedding commands (§16.8.2) 16-15



- "~f" format directive 5-14
- f lisp editor command 16-11
- (f) lisp editor form 16-11
- (f=) lisp editor form 16-11
- (fact) function 3-9
- (fake) function 6-3
- (fasl) function 5-3
- ?fast tpl command 13-3
- (fboundp) function 2-18
- (fclosure) function 8-15
- (fclosure-alist) function 8-15
- (fclosure-function) function 8-15
- (fclosure-list) function 8-15
- (fclosurep) function 2-10, 8-15
- Fclosures (§8.4.3) 8-14
- (feature-present) function 6-10
- \*fexpr, compiler declaration 12-2
- (ffasl) function 18-3
- File subsystem implementation notes (§13.7) 13-14
- file-backup-append tpl variable 13-14
- file-backup-prepend tpl variable 13-14
- ?filein tpl command 13-12
- Filein notes (§13.7.1) 13-14
- (fileopen) function 5-3
- ?fileout tpl command 13-13
- Fileout notes (§13.7.2) 13-14
- (filepos) function 5-3
- (filestat) function 5-3
- (filestat-atime) function 5-4
- (filestat-ctime) function 5-4
- (filestat-dev) function 5-4
- (filestat-gid) function 5-4
- (filestat-ino) function 5-4
- (filestat-mode) function 5-4
- (filestat-mtime) function 5-4
- (filestat-nlink) function 5-4
- (filestat-rdev) function 5-4
- (filestat-size) function 5-4
- (filestat-type) function 5-4
- (filestat-uid) function 5-4
- ?filestatus tpl command 13-13
- (fillarray) function 2-29
- (find-all-symbols) function 17-9
- (find-package) function 17-5
- (find-symbol) function 17-7
- (fix) function 3-9
- Fixnum (§1.1.5) 1-3
- :fixnum defstruct type 14-14
- Fixnum functions (§12.9) 12-9
- fixnum-block use in arrays 9-4
- (fixp) function 3-3
- (flatc) function 5-4
- (flatsize) function 5-4
- Flavor families (§19.10) 19-32
- Flavor functions (§19.8) 19-16
- (flavor-allows-init-keyword-p) function 19-23
- (float) function 3-9
- float-format constant B-3
- (floatp) function 3-3
- (float-to-double) function 18-16
- Flonum (§1.1.6) 1-3
- flonum-block use in arrays 9-4
- foo trace argument form 11-1
- (foo break) trace argument form 11-1
- (foo evalin expression) trace argument form 11-2
- (foo evalinout expression) trace argument form 11-2
- (foo evalout expression) trace argument form 11-2
- (foo evfcn evfunc) trace argument form 11-3
- (foo if expression) trace argument form 11-2
- (foo ifnot expression) trace argument form 11-2
- (foo lprint) trace argument form 11-2
- (foo printargs prfunc) trace argument form 11-3
- (foo printres prfunc) trace argument form 11-3
- (foo traceenter tefunc) trace argument form 11-3
- (foo traceexit txfunc) trace argument form 11-3
- Foreign-function disciplines (§18.3) 18-4
- (fork) function 6-3
- Format (§5.3) 5-11
- (format) function 5-12
- Format argument processing (§5.3.2) 5-17
- Format directives (§5.3.1) 5-13
- Fortran and Pascal (§18.6) 18-18
- (freturn) function 4-13
- (fs) lisp editor form 16-11



- (fseek) function 5-5
- (funcall) function 19-22, 4-13
- funcallhook* constant B-3
- (funcallhook) function 4-14
- :funcall-inside-yourself* flavor keyword option 19-34
- (funcall-self) function 19-22
- (function) function 4-14
- "function" function discipline 18-4
- Functional objects table (§8.5) 8-16
- Functions (§4.2) 4-1
  - (§5.2) 5-2
  - (§7.7) 7-12
  - (§8.2) 8-1
- "~g" format directive 5-18
- g step command 15-2
- (gc) function 6-4
- (gcafter) function 6-4
- (gcbefore) function 6-4
- \$gccount\$* constant B-1
- gcdisable* constant B-3
- \$gcprint* constant B-1
- gcstrings*; Status variable 6-4
- General arrays (§9.1) 9-1
- General strategy of the compiler (§12.1) 12-1
- Generic operations (§19.4) 19-6
- Generic operations in LISP (§19.5) 19-7
- (gensym) function 2-14
- (gentemp) function 2-14
- (get) function 2-33
- (getaccess) function 2-28
- (getaddress) function 18-3, 2-35
- (getaux) function 2-28
- (getchar) function 2-18
- (getcharn) function 2-18
- (getd) function 2-18
- (getdata) function 2-28
- (getdelta) function 2-28
- (getdisc) function 2-31
- (getentry) function 2-31
- (getenv) function 6-4
- :get-handler-for* flavor keyword option 19-33
- (get-handler-for) function 19-23
- (gethash) function 2-25
- (getl) function 2-33
- (getlength) function 2-28
- (get\_pname) function 2-18
- (getsyntax) function 7-13
- :gettable-instance-variables* flavor keyword option 19-24
- (go) function 4-14
- (greaterp) function 3-4
- H fixit command 15-6
- (haipart) function 3-5
- Hash table functions (§2.5.1) 2-24
- Hash tables (§1.1.16) 1-5
  - (§2.5) 2-24
- (hash-table-count) function 2-26
- (hash-table-p) function 2-10
- (haulong) function 3-5
- HELP fixit command 15-6
- (help) function 4-14
- ?help tpl command 13-2
- ?his[tory] tpl command 13-3
- How to use tpl (§13.3) 13-1
- Hunk (§1.1.13) 1-5
- :hunk* defstruct type 14-13
- (hunk) function 2-30
- Hunk accessor (§2.7.2) 2-30
- Hunk creation (§2.7.1) 2-30
- Hunk manipulators (§2.7.3) 2-31
- (hunkp) function 2-9
- Hunks (§2.7) 2-30
- (hunksize) function 2-31
- (hunk-to-list) function 2-30
- (i) lisp editor form 16-19
- ibase* constant B-3
- (if) function 4-15, 4-5
  - lisp editor form 16-20
- Immediate vectors (§9.6) 9-5
- Implementation of flavors (§19.13) 19-42
- (implode) function 2-12
- (implodes) function 2-12
- (import) function 17-8
- :include* defstruct keyword 14-6
- (include) function 6-4
- Include and includef (§12.3.2.4) 12-3
- :included-flavors* flavor keyword option



- 19-27
- (**include**) function 6-4
- (**include-if**) function 6-5
- (**include-if**) function 6-4
- (**infile**) function 5-5
- Infix (§7.6.1.3) 7-11
- :init** flavor keyword option 19-19
- :initable-instance-variables** flavor keyword option 19-24
- :initial-offset** defstruct keyword 14-9
- :init-keywords** flavor keyword option 19-25
- (**initsym**) function 2-15
- (**in-package**) function 17-5
- Input/output (Chapter 5) 5-1
- (**insert**) function 2-6
  - lisp editor form 16-14, 16-15
- Installation guide (Prependix i) i-1
- (**instancep**) function 19-19
- (**instantiate-flavor**) function 19-18
- "integer-function" function discipline 18-5
- (**integer-length**) function 3-5
- Integers and real numbers (§18.4.1) 18-6
- Interaction with step (§15.5.2) 15-7
- Interaction with trace (§15.5.1) 15-7
- (**intern**) function 17-6, 2-13
- Internal order (§9.5.3) 9-5
- Interrupt processing (§10.6) 10-3
- Introduction (§13.1) 13-1
  - (§14.1) 14-1
  - (§16.1) 16-1
  - (§17.1) 17-1
  - (§19.1) 19-1
  - (§4.1) 4-1
  - (§5.1) 5-1
  - (§7.1) 7-1
  - (Chapter 1) 1-1
- :inverse-list** flavor keyword option 19-38, 19-41
- (**\*invmod**) function 3-3
- Invocations (§7.6.2) 7-12
- Iteration strings (§5.3.3.2) 5-19
- 
- (**keywordp**) function 2-9
- (**kwote**) function 2-36
- 
- ?l tpl stepper command 13-5
- lambda*; compiler declaration 12-2
- (**last**) function 2-4
- (**lc**) lisp editor form 16-12
- (**lcl**) lisp editor form 16-12
- (**lconc**) function 2-34
- ?ld tpl command 13-3
- (**ldiff**) function 2-5
- \$ldprint* constant B-1
- :left-margin** wide-print-list keyword 5-11
- Length (§9.2.4) 9-2
- (**length**) function 2-3
- (**lessp**) function 3-4
- (**let**) function 4-15
- (**let\***) function 4-16
- (**let-closed**) function 8-14
- lexpr*; compiler declaration 12-2
- \*lexpr*; compiler declaration 12-2
- (**lexpr-funcall**) function 4-16
- (**lexpr-funcall-self**) function 19-22
- (**lexpr-send**) function 19-8
- (**lexpr-send-self**) function 19-22
- (**li**) lisp editor form 16-16
- :libdir** change-library-directory keyword E-2
- linel* constant B-3
- Lisp data (§18.4.6.6) 18-15
- LISP in a pipeline (§18.7.1) 18-22
- lisp-library-directory* constant B-3
- lisp-object-directory* constant B-3
- Lispval (§1.1.1) 1-1
- List (§1.1.3) 1-2
- :list** defstruct type 14-13
- :list\*** defstruct type 14-13
- :list** flavor keyword option 19-38, 19-41
- (**list**) function 2-1
- (**list\***) function 2-1
- List accessing (§2.1.3) 2-4
- List creation (§2.1.1) 2-1
- List manipulation (§2.1.4) 2-5
- List modification commands (§16.9) 16-16
- List predicates (§2.1.2) 2-3
- (**list-all-packages**) function 17-6
- (**listarray**) function 2-28
- (**listify**) function 4-16
- (**listp**) function 2-3, 2-9
- Lists (§18.4.4) 18-8
  - (§2.1) 2-1



- (list-to-bignum) function 2-3
- Liszt, the LISP compiler (Chapter 12) 12-1
- (litarom) function 2-9
- (lo) lisp editor form 16-17
- (load) function 5-5
- Loading C functions (§18.2) 18-2
- load-most-recent* constant B-3
- localf*, compiler declaration 12-2
- Location specifications (§16.7) 16-12
- (log) function 3-9
- (logand) function 3-7
- (logandc1) function 3-7
- (logandc2) function 3-7
- (logbitp) function 3-7
- (logcount) function 3-8
- (logeqv) function 3-7
- (logior) function 3-7
- (logiorc1) function 3-8
- (logiorc2) function 3-8
- (lognand) function 3-8
- (lognor) function 3-8
- (lognot) function 3-7
- (logtest) function 3-8
- (logxor) function 3-7
- (lp) lisp editor form 16-20
- (lpq) lisp editor form 16-20
- (lsh) function 3-6
- (lsubst) function 2-7
- Lxref (Appendix D) D-1
  
- (m) lisp editor form 16-21
- ?m tpl stepper command 13-6
- Macro expansion (§12.3.1) 12-1
- Macro forms (§8.3.1) 8-6
- (macroexpand) function 2-35
- Macros (§8.3) 8-6
- (makefn) lisp editor form 16-23
- (make-hash-table) function 2-24
- (make-instance) function 19-18
- (make-name) function 14-15
- (make-package) function 17-5
- (makereadtable) function 5-6
- (makhunk) function 2-30
- (\*makhunk) function 2-30
- (maknam) function 2-12
- (maknum) function 6-5
- (makunbound) function 2-20
  
- (map) function 4-16
- (mapc) function 4-16
- (mapcan) function 4-16
- (mapcar) function 4-16
- (mapcon) function 4-16
- (maphash) function 2-25
- (maplist) function 4-17
- mark lisp editor command 16-7
- (marray) function 2-27
- (max) function 3-9
- (mbd) lisp editor form 16-15
- (member) function 2-12
- Memory allocation (§18.5.2) 18-17
- (memq) function 2-12
- (merge) function 2-6
- Method combination (§19.12) 19-34
- :method-combination** flavor keyword
  - option 19-30
- (mfunction) function 4-17
- (min) function 3-9
- (minus) function 3-2
- (minusp) function 3-4
- Miscellaneous (§2.4.5) 2-23
- Miscellaneous editor commands (§16.14) 16-22
- Mixing flavors (§19.7) 19-12
- :mixture** flavor keyword option 19-30
- (mod) function 3-9
- (\*mod) function 3-9
- Modularity (§19.3) 19-3
- Modules (§17.7) 17-9
- \*modules\* variable 17-10
- (move) lisp editor form 16-15
- Move and copy commands (§16.8.3) 16-15
- (msg) function 5-6
- Multiple error levels (§15.5.3) 15-8
- Multiple value returns (§4.3) 4-20
- (multiple-value-bind) function 4-21
- (multiple-value-call) function 4-20
- (multiple-value-list) function 4-21
- (multiple-value-prog1) function 4-21
- (multiple-value-setq) function 4-21
- multiple-values-limit* constant B-4
- (mv) lisp editor form 16-16



- n lisp editor command 16-8
- n lisp editor command 16-8
- (-n) lisp editor form 16-13
- (n) lisp editor form 16-13, 16-14
- n step command 15-2
- :named** defstruct keyword 14-6, 14-8
- :named-array** defstruct type 14-13
- :named-hunk** defstruct type 14-13
- :named-list** defstruct type 14-13
- :named-vector** defstruct type 14-14
- :nconc** flavor keyword option 19-38, 19-41
- (nconc) function 2-7
- (ncons) function 2-1
- (neq) function 2-10
- (nequal) function 2-10
- (newsym) function 2-15
- (new-vector) function 2-21
- (new-vectori-byte) function 2-21
- (new-vectori-double) function 2-21
- (new-vectori-float) function 2-21
- (new-vectori-long) function 2-21
- (new-vectori-word) function 2-21
- nex lisp editor command 16-10
- (nex) lisp editor form 16-10
- nil constant B-4
- nil lisp editor command 16-22
- nlambd; compiler declaration 12-2
- no method type** flavor keyword option 19-40
- Normal (§7.6.1.1) 7-10
- (not) function 2-11
- :no-vanilla-flavor** flavor keyword option 19-28
- (nreconc) function 2-8
- (nreverse) function 2-8
- (nstring-capitalize) function 2-17
- (nstring-downcase) function 2-17
- (nstring-upcase) function 2-17
- (nth) function 2-4
  - lisp editor form 16-9
- (nthcdr) function 2-4
- (nthchar) function 2-18
- (nthelem) function 2-4
- (null) function 2-11
- (numberp) function 3-3
- (numbp) function 3-3
- (nwrtn) function 5-6
- !nx lisp editor command 16-9

- nx lisp editor command 16-9
- (nx) lisp editor form 16-9
- #O character macro 8-10
- #o character macro 8-10
- "~o" format directive 5-14
- :objdir** change-library-directory keyword E-2
- Objects (§19.2) 19-1
- Objects, message-passing, and flavors (Chapter 19) 19-1
- (oblist) function 4-17
- (oddp) function 3-3
- OK fixit command 15-6
- ok lisp editor command 16-22
- (oldsym) function 2-15
- (onep) function 3-3
- :operation-handled-p** flavor keyword option 19-33
- (opval) function 6-5
- :or** flavor keyword option 19-38, 19-40, 19-41
- (or) function 4-17
- Order of definition (§19.13.1) 19-43
- :ordered-instance-variables** flavor keyword option 19-28
- (orf) lisp editor form 16-12
- (orr) lisp editor form 16-20
- Other (§1.1.17) 1-5
- Other caveats (§14.3.1.1.4) 14-16
- Other forms (§12.3.2.6) 12-4
- Other functions (§3.6) 3-8
- Other trace package functions (§11.3) 11-5
- (outfile) function 5-6
- :outside-accessible-instance-variables** flavor keyword option 19-29
- Overhead of stepping (§15.3) 15-3
- :override** flavor keyword option 19-40

- P fixit command 15-5
- "~p" format directive 5-16
- /p lisp editor command 16-7
- p lisp editor command 16-7
- (p) lisp editor form 16-7, 16-8
- p step command 15-2
- ?P tpl stepper command 13-5



- ?p tpl stepper command 13-5
- Package (§1.1.14) 1-5
- \*package\* constant B-4
- \*package\* variable 17-5
- Package names (§17.2) 17-2
- Package system functions and variables (§17.6) 17-4
- (package-name) function 17-6
- (package-nicknames) function 17-6
- (packagep) function 2-9
- Packages (Chapter 17) 17-1
- (package-shadowing-symbols) function 17-6
- (package-used-by-list) function 17-6
- (package-use-list) function 17-6
- Parentheses-moving commands (§16.9.1) 16-16
- Passing C structures by value (§18.4.5) 18-10
- :pass-on flavor keyword option 19-38, 19-41
- (patom) function 5-7
- Pattern and search commands (§16.6) 16-10
- Pattern specification (§16.6.1) 16-10
- Pipes (§18.7) 18-22
- piport constant B-4
- (plist) function 2-33
- (plus) function 3-1
- (plusp) function 3-3
- (pntlen) function 5-7
- Pointers to structures (§18.4.6.5) 18-15
- (pop) function 4-17
- ?pop tpl command 13-4
- poport constant B-4
- Port (§1.1.9) 1-3
- :port wide-print-list keyword 5-11
- (portp) function 5-7
- PP fixit command 15-5
- (pp) function 5-7
- pp lisp editor command 16-8
- pp\* lisp editor command 16-8
- ?pp tpl command 13-4
- (pp-form) function 5-8
- ?ppp tpl stepper command 13-5
- ?PPP tpl stepper command 13-6
- Predicates (§2.2) 2-8
- (§3.2) 3-3
- Preface (Prependix p) p-1
- (primep) function 3-4
- (princ) function 5-7
- prinlength constant B-4
- prinlevel constant B-4
- :print defstruct keyword 14-12
- (print) function 5-7
- :print-function defstruct keyword 14-12
- Printing commands (§16.4) 16-7
- :print-self flavor keyword option 19-33
- (probet) function 5-8
- (\*process) function 6-5
- (process) function 6-6
- (\*process-receive) function 6-6
- (\*process-send) function 6-6
- (product) function 3-2
- (prof-end) function 6-13
- (prof-report) function 6-13
- (prof-start) function 6-13
- (prog) function 4-18
- (prog1) function 4-18
- (prog2) function 4-18
- Progn (§12.3.2.3) 12-3
- :progn flavor keyword option 19-37, 19-41
- (progn) function 4-18
- Program forms (Chapter 8) 8-1
- (progv) function 4-18
- Property (§9.5.2) 9-5
- Property list (§2.9.2) 2-32
- Property list operations (§19.14) 19-45
- Protection against reevaluation (§8.3.3) 8-7
- (provide) function 17-10
- ?prt tpl command 13-3
- (ptime) function 6-6
- ptport constant B-4
- (ptr) function 2-36
- (purcopy) function 4-18
- Pure literals (§12.7) 12-8
- (purep) function 4-18
- (push) function 4-19
- (pushnew) function 4-19
- (putaccess) function 2-29
- (putaux) function 2-29
- (putd) function 4-19
- (putdata) function 2-29
- (putdelta) function 2-29
- (putdisc) function 2-31
- (putlength) function 2-29
- (putprop) function 2-33



- "~q" format directive 5-17
- q step command 15-2
- (qualify-aexplode) function 2-21
- (qualify-aexplodec) function 2-21
- (qualify-aexploden) function 2-21
- (qualify-always-escape-aexploden) function 2-21
- (qualify-escape-aexploden) function 2-21
- (qualify-escape-exploden) function 2-20
- (qualify-explode) function 2-20
- (qualify-explodec) function 2-20
- (qualify-exploden) function 2-20
- (\*quo) function 3-2
- (quote!) function 2-2
- (quote) function 2-36
- (quotient) function 3-2
  
- "~r" format directive 5-14
- (r) lisp editor form 16-14
- (random) function 3-9
- Random functions (§2.10) 2-35
- (rassq) function 2-31
- (ratom) function 5-8
- (read) function 5-8
- Read time evaluation (§8.3.5.2) 8-12
- (readc) function 5-9
- (readdir) function 5-9
- Reader operations (§7.3) 7-2
- (readline) function 5-9
- (readllst) function 5-9
- readtable constant B-4
- "real-function" function discipline 18-5
- (recompile-flavor) function 19-22
- Reconfiguring lisp (Appendix E) E-1
- REDO fixit command 15-6
- :rehash-size make-hash-table keyword 2-24
- :rehash-threshold make-hash-table keyword 2-25
- Release notes (Prependix r) r-1
- (remainder) function 3-9
- (rematom) function 2-13
- ?rem-function tpl command 13-13
- (remhash) function 2-25
- (remob) function 2-13
- (remove) function 2-6
- (removeaddress) function 18-4, 5-9
- (remprop) function 2-33
- ?rem-prop tpl command 13-13
- (remq) function 2-6
- (remsym) function 2-16
- ?rem-var tpl command 13-13
- (rename-package) function 17-6
- repack lisp editor command 16-23
- (repack) lisp editor form 16-23
- (replace) function 2-36
- lisp editor form 16-15
- (require) function 17-10
- :required-flavors flavor keyword option 19-26
- :required-init-keywords flavor keyword option 19-26
- :required-instance-variables flavor keyword option 19-26
- :required-methods flavor keyword option 19-26
- (reset) function 6-6
- ?re[set] tpl command 13-3
- (resetio) function 5-9
- ?ret tpl command 13-4
- RETURN fixit command 15-6
- "~RETURN" format directive 5-16
- (return) function 4-19
- RETURN step command 15-1
- tpl stepper command 13-6
- (reverse) function 2-8
- (ri) lisp editor form 16-17
- (ro) lisp editor form 16-17
- (rot) function 3-7
- (rplaca) function 2-5
- (rplacd) function 2-5
- (rplacx) function 2-30
- (\*rplacx) function 2-30
- (\*rset) function 6-6
- Running the compiler (§12.2) 12-1
- :run-time-alternatives flavor keyword option 19-30
  
- #S character macro 8-10
- #s character macro 8-10
- "~s" format directive 5-14
- (s) lisp editor form 16-23
- Sample sessions with tpl (§13.5) 13-7
- (sassoc) function 2-32



- (**sassq**) macro 2-32
- ?**sc** tpl stepper command 13-5
- (**scons**) function 2-36
- Scope of attention (§16.5) 16-8
- Search commands (§16.6.2) 16-11
- (**second**) lisp editor form 16-13
- (**segment**) function 6-7
- Selectively turning on stepping (§15.2.1) 15-2
- (**selectq**) function 4-19
- self variable 19-21
- Semantics (§18.4.6.3) 18-14
- (**send**) function 19-22, 19-8
- :**send-if-handles** flavor keyword option 19-33
- (**send-self**) function 19-22
- (**set**) function 2-19
- (**setarg**) function 4-19
- (**setf**) function 14-2
- Setf and defsetf (§14.2) 14-1
- (**set-in-closure**) function 8-14
- (**set-in-fclosure**) function 8-15
- (**set-in-instance**) function 19-23
- (**setplist**) function 2-19, 2-33
- (**setq**) function 2-19
- (**setsyntax**) function 7-12
- :**settable-instance-variables** flavor keyword option 19-24
- (**shadow**) function 17-8
- (**shadowing-import**) function 17-8
- Sharp sign character macros (§8.3.5) 8-9
- (**shell**) function 6-7
- (**showstack**) function 6-7
- (**si:flavor-allowed-init-keywords** 'flavor-name) function 19-23
- (**signal**) function 6-7
- (**signp**) function 2-10
- Simple arithmetic functions (§3.1) 3-1
- Simple use of flavors (§19.6) 19-9
- Simple use of stepping (§15.1) 15-1
- (**sin**) function 3-5
- (**si:property-list-mixin**) function 19-45, 19-46
- si:vanilla-flavor** flavor keyword option 19-33
- Size (§9.5.1) 9-4
- :**size** make-hash-table keyword 2-24
- :**size-macro** defstruct keyword 14-9
- (**sizeof**) function 6-7
- :**size-symbol** defstruct keyword 14-9
- (**sload**) function 5-9
- (**small-segment**) function 6-7
- ?**soff** tpl stepper command 13-5
- (**sort**) function 2-37
- (**sortcar**) function 2-37
- ?**sp** tpl stepper command 13-6
- "**SPACE**" format directive 5-16
- SPACE RETURN** tpl stepper command 13-6
- special*; compiler declaration 12-2
- Special forms (§12.3) 12-1
- Special functions (Chapter 4) 4-1
- Special symbols (Appendix B) B-1
- :**special-instance-variables** flavor keyword option 19-25
- Splicing (§7.6.1.2) 7-11
- (**sprintf**) function 5-9
- (**sqrt**) function 3-9
- (**sstatus**) function 6-8
- (**sstatus appendmap**) function 6-8
- (**sstatus automatic-reset**) function 6-8
- (**sstatus chainatom**) function 6-8
- (**sstatus dumpcore**) function 6-8
- (**sstatus evalhook**) function 6-8
- (**sstatus feature**) function 6-8
- (**sstatus nofeature**) function 6-9
- (**sstatus translink**) function 6-9
- (**sstatus uctoic**) function 6-9
- ?**state** tpl command 13-3
- (**status**) function 6-9
- (**status ctime**) function 6-9
- (**status feature**) function 6-9
- (**status features**) function 6-9
- (**status isatty**) function 6-10
- (**status localtime**) function 6-10
- (**status syntax**) function 6-10
- (**status undeffunc**) function 6-11
- (**status version**) function 6-11
- STEP** fixit command 15-6
- (**step**) function 15-1
- ?**step** tpl stepper command 13-5
- Stepper commands (§13.3.1) 13-5
- Stepping with breakpoints (§15.2.2) 15-3
- (**sticky-bignum-leftshift**) function 3-5
- stop** lisp editor command 16-22
- (**store**) function 2-29
- (**strcat**) function 2-12



- String (§1.1.8) 1-3
- (string=) function 2-16
- (string<) function 2-16
- (string<=) function 2-16
- (string=) function 2-16
- (string>) function 2-16
- (string>=) function 2-16
- (string) function 2-19
- String and symbol predicates (§2.3.2) 2-16
- (string-capitalize) function 2-17
- (string-downcase) function 2-17
- (string-length) function 2-17
- "~<[stringn" format directive 5-18
- (stringp) function 2-9
- Strings and symbols (§18.4.2) 18-6
- (string-upcase) function 2-17
- Structure modification commands (§16.8) 16-13
  - (§16.8.1) 16-13
- Structured data types (Chapter 14) 14-1
- Structures (§2.9) 2-31
- (sub1) function 3-2
- (sublis) function 2-32
- (subpair) function 2-7
- Subparts of an array object (§9.2) 9-2
- "subroutine" function discipline 18-4
- (subst) function 2-7
- (substring) function 2-18
- (substringn) function 2-18
- (substringp) function 2-19
- (sum) function 3-1
- Supplements (Supplement S) S-1
- (sw) lisp editor form 16-14
- (sxhash) function 2-27
- Symbol (§1.1.2) 1-1
- Symbol and string accessing (§2.3.3) 2-17
- Symbol and string creation (§2.3.1) 2-12
- Symbol and string manipulation (§2.3.4) 2-19
- Symbol generating facility (§2.3.1.1) 2-14
- (symbol-function) function 2-18
- (symbol-name) function 2-18
- (symbolp) function 2-9
- (symbol-package) function 2-18
- (symbol-plist) function 2-33
- Symbols and strings (§2.3) 2-12
- (symbol-value) function 2-17
- (symeval) function 2-17
- (symeval-in-closure) function 8-14
- (symeval-in-fclosure) function 8-15
- (symeval-in-instance) function 19-23
- (symstat) function 2-16
- Syntax (§18.4.6.2) 18-13
- Syntax classes (§7.2) 7-1
  - (§7.5) 7-6
- (sys:access) function 6-12
- (sys:chmod) function 6-11
- (sys:getpid) function 6-12
- (sys:getpwnam) function 6-12
- (sys:getpwuid) function 6-12
- (sys:getuid) function 6-12
- (sys:link) function 6-12
- System functions (§6.1) 6-1
  - (Chapter 6) 6-1
- (sys:time) function 6-12
- (sys:unlink) function 6-12
- t constant B-5
- "~t" format directive 5-16
- (tab) function 5-10
- (tailp) function 2-3
- (tconc) function 2-34
- Tconc structure (§2.9.3) 2-34
- (terpr) function 5-10
- (terpri) function 5-10
- test lisp editor command 16-18
- :test make-hash-table keyword 2-24
- "~<text3" format directive 5-21
- The anatomy of an error (§10.2) 10-1
- The backquote character macro (§8.3.4) 8-8
- The C program (§18.5) 18-16
- The edit chain (§16.3.1) 16-6
- The file subsystem (§13.6) 13-12
- The fixit debugger (§15.5) 15-4
- The foreign-function interface (Chapter 18) 18-1
- The garbage collector (Appendix C) C-1
- The LISP editor (Chapter 16) 16-1
- The Lisp Profiler (§6.2) 6-13
- The LISP reader (Chapter 7) 7-1
- The LISP stepper and fixit (Chapter 15) 15-1
- The Lister trace package (Chapter 11) 11-1
- The MacLISP-compatible array package (§9.3) 9-3
- The perils of pipes (§18.7.3) 18-23



- The top level (Chapter 13) 13-1
- The trace function (§11.1) 11-1
- (third) lisp editor form 16-13
- (**\*throw**) function 4-19
- (throw) function 4-19
- (tilde-expand) function 5-10
- (times) function 3-2
- (time-string) function 6-12
- tl lisp editor command 16-23
- To-and-thru commands (§16.9.2) 16-17
- TOP fixit command 15-5
- :**top-level** change-library-directory keyword E-2
- top-level constant B-5
- (top-level) function 6-13
- top-level-eofs tpl special symbol 13-6
- top-level-init tpl special symbol 13-6
- top-level-print tpl special symbol 13-7
- top-level-prompt tpl special symbol 13-6
- Tpl special symbols (§13.4) 13-6
- tpl-history-show tpl special symbol 13-7
- tpl-number-prompt tpl special symbol 13-7
- tpl-prinlength tpl special symbol 13-7
- tpl-prinlevel tpl special symbol 13-7
- ?tr tpl command 13-3
- (trace) function 11-1
- Trace function argument forms (§11.1.1) 11-1
- Trace special variables (§11.2) 11-4
- (traceargs) function 11-4
- (tracedump) function 11-5
- trace-prinlength constant 11-4
- trace-prinlevel constant 11-4
- Transfer tables (§12.8) 12-8
- Translating strings to symbols (§17.3) 17-3
- :**tree** defstruct type 14-13
- Trigonometric functions (§3.3) 3-4
- "~:[true]" format directive 5-18
- (true-name) function 5-10
- tty lisp editor command 16-22
- Tutorial (§16.2) 16-1
- (tyi) function 5-10
- (typepeek) function 5-10
- (tyo) function 5-10
- :**type** defstruct keyword 14-8
- (type) function 2-10
- (typep) function 19-16, 2-10
- Types (§7.6.1) 7-10
- Types of structures (§14.3.1.1.1) 14-13
- U fixit command 15-6
- (uconcat) function 2-12
- uctolc; Status variable 6-9
- unblock lisp editor command 16-18
- (undefflavor) function 19-21
- (undefmethod) function 19-21
- !undo lisp editor command 16-18
- undo lisp editor command 16-18
- Undoing commands (§16.10) 16-18
- (unexport) function 17-7
- (unintern) function 17-7
- (unless) function 4-5
- unspecial; compiler declaration 12-2
- ?untr tpl command 13-3
- (untrace) function 11-5
- (untyi) function 5-10
- (unuse-package) function 17-8
- (unwind-protect) function 4-19
- UP fixit command 15-6
- up lisp editor command 16-8
- ?up tpl command 13-4
- Useful functions (§8.4.2) 8-14
- (use-package) function 17-8
- usermacros variable 16-22
- user-top-level constant B-5
- tpl special symbol 13-6
- Using defstruct (§14.3.1.1) 14-13
- Using subprocesses (§18.7.2) 18-23
- Using the compiler (§12.4) 12-4
- Using the stepper (§13.5.1) 13-10
- Using trace options (§11.1.2) 11-3
- Valid function objects (§8.1) 8-1
- Value (§1.1.12) 1-4
- (valuep) function 2-9
- (values) function 4-20
- (values-list) function 4-20
- Vanilla flavor (§19.11) 19-32
- vcharacter syntax class 7-7
- Vector (§1.1.10) 1-4
- :**vector** defstruct type 14-14
- (vector) function 2-21
- Vector creation (§2.4.1) 2-21
- Vector modification (§2.4.4) 2-23



Vector predicate (§2.4.2) 2-22  
 Vector reference (§2.4.3) 2-22  
*vector-bounds-check*; compiler declaration 12-3  
*"vector-c-function"* function discipline 18-5  
*(vectori-byte)* function 2-22  
*(vectori-byte-to-string)* function 2-23  
*(vectori-double)* function 2-22  
*(vectori-float)* function 2-22  
*(vectori-long)* function 2-22  
*(vectorip)* function 2-9  
*(vectori-word)* function 2-22  
*(vectorp)* function 2-22, 2-9  
 Vectors (§2.4) 2-21  
   (§9.4) 9-4  
 Vectors and arrays (§18.4.3) 18-6  
*vescape* syntax class 7-8  
*(vget)* function 2-22  
*villegal* syntax class 7-10  
*vinfix-macro* syntax class 7-9  
*vleft-bracket* syntax class 7-7  
*vleft-paren* syntax class 7-7  
*vmacro* syntax class 7-9  
*vnumber* syntax class 7-7  
*"void-c-function"* function discipline 18-5  
*vpackage* syntax class 7-7  
*vperiod* syntax class 7-8  
*(vprop)* function 2-22  
*(vputprop)* function 2-23  
*(vref)* function 2-22  
*(vrefi-byte)* function 2-22  
*(vrefi-double)* function 2-22  
*(vrefi-float)* function 2-22  
*(vrefi-long)* function 2-22  
*(vrefi-word)* function 2-22  
*vright-bracket* syntax class 7-7  
*vright-paren* syntax class 7-7  
*vseparator* syntax class 7-8  
*(vset)* function 2-23  
*(vseti-byte)* function 2-23  
*(vseti-double)* function 2-23  
*(vseti-float)* function 2-23  
*(vseti-long)* function 2-23  
*(vseti-word)* function 2-23  
*(vsetprop)* function 2-23  
*vsign* syntax class 7-7  
*vsingle-character-symbol* syntax class 7-9  
*vsingle-infix-macro* syntax class 7-10

*vsingle-macro* syntax class 7-9  
*vsingle-quote* syntax class 7-8  
*vsingle-splicing-macro* syntax class 7-9  
*(vsize)* function 2-22  
*(vsize-byte)* function 2-23  
*(vsize-double)* function 2-23  
*(vsize-float)* function 2-23  
*(vsize-word)* function 2-23  
*vsplicing-macro* syntax class 7-9  
*vstring-delimiter* syntax class 7-8  
*vsymbol-delimiter* syntax class 7-8

*`w* constant B-2  
*(wait)* function 6-13  
*(when)* function 4-5  
 WHERE fixit command 15-6  
 ?whichfile tpl command 13-13  
*:which-operations* flavor keyword option 19-33  
*(wide-print-list)* function 5-11  
*:wrapper* flavor keyword option 19-41

#X character macro 8-11  
 #x character macro 8-11  
 "~x" format directive 5-16  
*(xcons)* function 2-1  
*(xtr)* lisp editor form 16-15

*(y-or-n-p)* function 5-11

*(zapline)* function 5-11  
*(zerop)* function 3-3  
 ?zo tpl command 13-4



# Supplements





# Supplements

This section of the *FRANZ LISP Reference Manual: Opus 43* contains supplements that apply to specific host operating-system and hardware configurations. If you received any supplements with your documentation kit, you may wish to insert them in this section for your convenience.





# Installation guide





# Installation guide

This section of the *FRANZ LISP Reference Manual: Opus 43* contains the installation guide that applies to your specific operating-system and hardware configuration. You may wish to insert the installation guide that you received with your distribution of FRANZ LISP in this section for your future convenience.





# Release notes





## Release notes

This section of the *FRANZ LISP Reference Manual: Opus 43* contains the release notes for FRANZ LISP. You may wish to insert the release notes that you received with your distribution of FRANZ LISP in this section for your future convenience.





■

The *FRANZ LISP Reference Manual* was printed on an Apple Laserwriter laser printer driven by Adobe Systems' Postscript. The manual was typeset using the Unix device-independent troff program, with tables preprocessed by tbl and equations by eqn. The Index, Contents, and Appendix A were generated automatically. The text is set in Times Roman and Helvetica. Examples are set in Courier.

■







